

# Open Research Online

---

The Open University's repository of research publications and other research outputs

## A Methodology for the Development of Recurrent Networks for Sequence Processing Tasks

### Thesis

#### How to cite:

Bradbury, David (1997). A Methodology for the Development of Recurrent Networks for Sequence Processing Tasks. PhD thesis The Open University.

For guidance on citations see [FAQs](#).

© 1997 David Bradbury



<https://creativecommons.org/licenses/by-nc-nd/4.0/>

Version: Version of Record

Link(s) to article on publisher's website:

<http://dx.doi.org/doi:10.21954/ou.ro.0000fe7a>

---

Copyright and Moral Rights for the articles on this site are retained by the individual authors and/or other copyright owners. For more information on Open Research Online's data [policy](#) on reuse of materials please consult the policies page.

---


[oro.open.ac.uk](http://oro.open.ac.uk)

# A Methodology for the Development of Recurrent Networks for Sequence Processing Tasks

David Bradbury

Thesis submitted in partial fulfillment of the  
requirements for Ph D.

September 1996

  
Date of submission: 5<sup>th</sup> September 1996  
Date of award: 31<sup>st</sup> October 1997

ProQuest Number:27701071

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 27701071

Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code  
Microform Edition © ProQuest LLC.

ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 – 1346

David Clifford Bradbury BA(Hons), M.Sc

A Methodology for the Development of Recurrent Networks for Sequence  
Processing Tasks

Submitted for the degree of PhD in Artificial Intelligence

Submitted 5 September 1996

## **Abstract**

Artificial neural networks are increasingly being used for dealing with real world applications. Many of these (e.g. speech recognition) are based on an ability to perform sequence processing. A class of artificial neural networks, known as recurrent networks, have architectures which incorporate feedback connections. This in turn allows the development of a memory mechanism to allow sequence processing to occur.

A large number of recurrent network models have been developed, together with modifications of existing architectures and learning rules. However there has been comparatively little effort made to compare the performance of these models relative to each other. Such comparative studies would show differences in performance between networks and allow an examination of what features of a network give rise to desirable behaviours such as faster learning and superior generalisation ability.

This thesis describes the results of a number of existing comparative studies and the results of new research. Three different recurrent networks, both in their original form and with modifications, are tested with four different sequence processing tasks. The results of this research clearly show that recurrent networks vary widely in terms of their performance and lead to a methodology based on the following conclusions:

- The architecture should be as simple as possible and as complex as necessary
- Learning rules where a change in connection strength is based on local variables only are superior to those which use non-local factors.
- Adaptive memory mechanisms are under exploited and are a particularly promising avenue for further research, particularly for those interested in their models having physiological validity.

Finally there are some speculations as to how these principles could be put into practice. Particularly the use of hybrid models using genetic algorithms for controlling the complexity of the network architecture.

## **Contents**

<b>Chapter One. Introduction</b>	<b>1</b>
1.1.1 Why Sequence Processing is Important	1
1.1.2 What Are Sequence processing problems?	1
1.1.3 Three Example Sequence processing Problems	2
1.2 Recurrent Networks: What are they and What are the Desirable Properties?	6
1.3 Overview of the Thesis	9
1.4 Aims of the Thesis	10
 <b>Chapter Two. Recurrent Networks: A Review of the Literature</b>	 <b>11</b>
2.1 A History of Sequence Processing Using Neural Networks	11
2.1.1 Back propagation Through Time (BPTT)	11
2.1.2 Using Shift Registers for Sequence Processing	12
2.1.3 First and Second Order Recurrent Networks	14
2.1.4 The Simple Recurrent Network (SRN)	17
2.1.5 Memory Neural Networks	25
2.1.6 Finite Impulse Response (FIR) Filters	28
2.1.7 The Real Time Recurrent Learning Model, and its variations	30
2.1.8 The Gamma Model	38
2.2 Issues of Convergence	41
2.3 Attempts at Classification	43
2.4 Other Comparative Studies	46
2.5 Conclusion	54

<b>Chapter Three. A Comparative Study of Three Recurrent Network Models</b>	<b>56</b>
3.1 Overview and Rationale	56
3.2 The Tasks	57
3.3 The Networks	64
3.4 Experimental Variables	67
3.5 Results	68
3.5.1 The Effects of Modifying Internal Parameters on Learning	68
3.5.2 Comparative Study Results	72
3.5.3 Statistical Analysis of Results	73
3.5.4 Continuous XOR With Two Step Delay	76
3.5.5 The Letter in Word Prediction Task	78
3.5.6 Learning a Finite State Grammar	81
3.5.7 Dollar to Swiss Franc Exchange Rate	83
3.6 Some Observations on the $\mu$ parameter	83
3.7 Discussion	85
3.7.1 Behaviour of Recurrent Networks in General	85
3.7.2 Why Different Recurrent Networks Behave Differently	88
3.8 What Next?	91
 <b>Chapter Four. Modifications to the RTRL Algorithm and Their Implications</b>	 <b>93</b>
4.1 Overview and Rationale	93
4.2 Modifying RTRL Network Architecture	94
4.2.1 Pruning Network Architectures	94
4.2.2 A Pruned RTRL Architecture	96
4.2.3 Randomly Pruning Connections During Learning	104
4.3 Modifying the RTRL Algorithm	107
4.3.1 Different Reset Intervals for Different Tasks?	112



4.3.2 Does Resetting $P_{ijk}$ Allow Increased Learning Rates?	115
4.4 Summary	122
Chapter Five. Summary and Conclusion	123
5.1 The Effect of Architecture on Learning	123
5.2 The Effect of the Learning Algorithm on Learning	127
5.3 What Does this Mean for Efficient Learning?	131
5.3.1 Adaptive Memory Mechanisms	132
5.4 Can we Improve Learning Without Changing the Network?	133
5.5 Summary	135
Chapter Six. Conclusions	137
6.1 An Aside on Genetic Algorithms	137
6.1.1 Genetic Algorithms and Recurrent Networks	139
6.2 Some Guiding Principles for Recurrent Network Development	143
6.3 Speculations (i): A New Metaphor for Recurrent Network Training	146
6.3.1 The New Metaphor and Amits Criteria	150
6.4 Speculations (ii): Future Research	152
References	155
Appendices	161
Appendix 1. The Gamma Model Learning Rule	161
Appendix 2. The Summation Function for Gamma Kernels	164
Appendix 3. The Original RTRL Algorithm	166
Appendix 4. A Modified Form of the RTRL Algorithm, For Use With Networks	



## **Chapter One. Introduction**

This chapter is divided into three sections: the first looks at why sequence processing is important. This point is illustrated with three quite distinct problems. The second section defines and describes recurrent artificial neural networks (henceforth referred to as recurrent networks). Finally there is an overview of the thesis.

### **1.1.1 Why sequence processing is important**

Artificial neural networks which consist of relatively simple units, organised into layers and linked by weighted connections (for a fuller description see Rumelhart and McClelland 1986) have been used to solve a wide range of problems. Many of these have been such that all the information that the network needs to solve the task is available at a single time step, for example face recognition. Many real world problems however are such that the information that the network needs to solve the task is only available over a number of time steps.

### **1.1.2 What are sequence processing problems?**

It is possible that a problem in which the information is received over time would not be a sequence processing problem because the order of the stimuli is not a factor in determining the output of the network. A sequence processing problem is one in which the order in which the information is presented is as important as the information itself in determining what the desired output of the neural network should be.

### **1.1.3 Three example sequence processing problems**

A look at three different problem domains will demonstrate that sequence processing problems are widespread and, therefore, the development of neural network models which can solve these problems quickly and efficiently is a worthwhile enterprise.

#### *Psychology*

One of the main areas of inquiry for cognitive psychology is the processes by which people perform sequence processing tasks such as speech recognition. An important part of the formulation of any valid psychological theory is the ability to build a model of the phenomena under investigation which accurately reflects the phenomena and has significant predictive abilities. Increasingly, neural networks are being used as a modelling tool.

Cleeremans and McClelland (1991) used a simple recurrent network (see section 1.2 for a definition of a simple recurrent network and section 2.1 for further details of this particular network) to model the way in which humans learn the structure of sequences. The human subjects were trained on sequential material based on a finite state grammar (see chapter three where the finite state grammar used in the first part of Cleeremans and McClelland's study is used to generate a data set).

Cleeremans and McClelland found that the simple recurrent network architecture was able to capture key aspects of both the learning and processing of event sequences. The initial version of the simple recurrent network was a poor model of human subject performance. However, modification of the activation and learning rules of the network, so that the former was a function not only of current input, but also a decaying trace of past inputs and the latter had two instead of one component (see section 2.1 for further details), led to a model which fitted the human subject data rather well.

## *Biology*

Underlying the psychological processes described above is the physiology of the brain. Artificial neural networks are far simpler than real neural networks such as the human brain in terms of the number of units in the network, the structure of these units and the way in which these units are organised. As well as underlying sequence processing abilities in general, there has also been speculation that the brain uses temporal processing to store information.

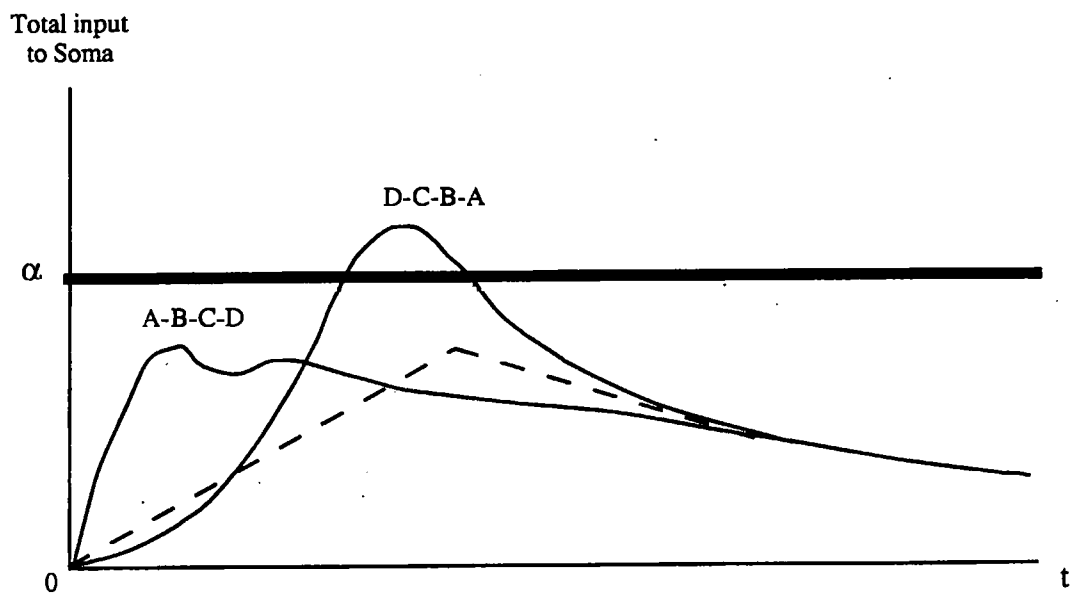
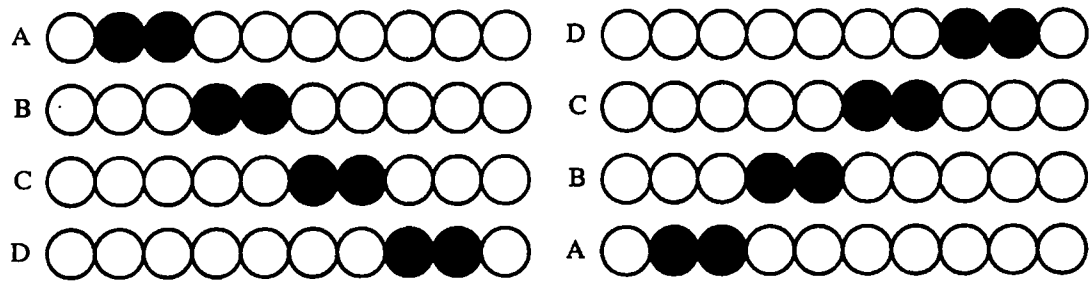
There is evidence that temporal processing takes place at the single neuron level. A typical "neuron" in an artificial neural network will simply sum its weighted inputs and pass the resulting value through a squashing function. A neuron in the brain, however, is a much more complex affair. Mel (1994) suggests that the dendritic tree of a single neuron may account for anything up to 99% of its total surface area and may have as many as 200,000 synaptic inputs. Across the brain as a whole, dendritic trees consume over 60% of the brain's energy. This complexity suggests that a dendritic tree is in fact a complex information processing device. One of the types of information processing abilities that dendritic trees are claimed to have is to act as a spatiotemporal filter<sup>1</sup>. Using compartmental modelling, Rall (1964) was the first to demonstrate that a passive dendritic branch can act as a filter that selects for specific temporal sequences.

Figure 1.1 demonstrates this principle. The peak of the voltage wave form is twice as large when the inputs to the neuron (simplified here as a ten compartment model) move closer to the soma during the course of the sequence. The difference is such that the sequence D-C-B-A causes the peak voltage to be greater than the firing

---

<sup>1</sup>An ability to perform spatiotemporal integration is one of four that researchers have proposed. The other three are: i) the existence of semi-independent processing sub units. ii) dendritic structure is influenced by the behaviour of the whole neuron or by its sub units. iii) Non-linear mechanisms mean that the dendritic tree can act as a logical network.

threshold  $\alpha$ . The dotted line represents the voltage if the components of the sequence are presented simultaneously. Note that in this example the neuron is broken down into a compartmental model. This involves breaking down the complex structure of the dendritic tree into sections (compartments), each of which consists of circuit elements which capture the electrical properties of the corresponding part of the dendritic tree.



*Figure 1.1.a: A ten component model of a dendritic branch demonstrates difference for different input sequences*



*Figure 1.1.b: Detail of the ten component model used in figure 1a. The shaded portion represents the soma. Non-shaded portions represent the dendritic tree.*

## *Engineering*

The term "engineering" represents a class of problems where the biological plausibility of the network is not as critical as it is in the two areas described above, so long as the task is performed efficiently. In these problem domains, neural networks often find themselves in competition with rule based or statistical techniques. Nevertheless much work has been done with neural networks, particularly in control problems (see for example Venugopal et al 1994). Neural networks have also been used in problem domains where the dynamics of a sequence need to be understood so that accurate predictions can be made concerning its future direction.

McCann and Kalman (1994) used a recurrent neural network architecture to predict turning points in the gold market. The architecture used was a simple recurrent network which also employed skip connections from the input to the output layer. With an appropriate trading strategy, it was found that a significant paper profit could be gained. The trading strategy used was as follows:

- IF we are in the market, and it has been the case that  $I < s$  for  $n$  consecutive days or more then SELL
- IF we are not in the market, and it has been the case that  $I > b$  for  $n$  consecutive days or more then BUY

In the above strategy,  $I$  is the network indicator,  $b$  is the buy threshold and  $s$  is the sell threshold. All these as well as the constant  $n$  were adjusted experimentally to find the optimum parameters.

## 1.2 Recurrent networks: What are they and what are the desirable properties?

Broadly speaking, recurrent networks are a class of neural network models that are distinctive because they have connections within layers and/or from a layer to a layer "lower down" the network (here a layer is lower if it is closer to the input layer). They are distinct from feed forward networks which only have connections from lower to higher layers and have no connections within layers. As we shall see in section 2.1, recurrent networks have become the architecture of choice when using neural networks for sequence processing tasks.

Recurrent networks are particularly suitable for sequence processing tasks because their feedback connections give rise to a capacity for storage of past activations. This acts as a short term memory structure, allowing the network to retain information concerning the order in which various inputs were presented to it. As we saw in section 1.1.2, memory for order is vital if sequence processing is to successfully undertaken.

The desirable properties of a recurrent network are identical to those of feed forward networks. To the casual observer it might seem that the only thing that a neural network has to do is to produce an output that is identical to the desired output. The truth however is more complex, as illustrated by the following quote:

"It should be first of all emphasised that a major task of any theory of neural networks is to produce *exceptional* input output relations. They have to be exceptional in that they should correspond, even if initially only in a metaphorical sense, to our intuitions about cognitive processes. Attractive features are *biological plausibility; associativity; parallel processing; emergent behaviour (cooperativity); freedom from homunculi; potential for abstraction*. Then, if any of these features are captured by the model, it has to prove robust to the type of disorder , fluctuations,



disruptions that we imagine the brain to be operating under." (Amit 1992 pp6. *Italics in original*).

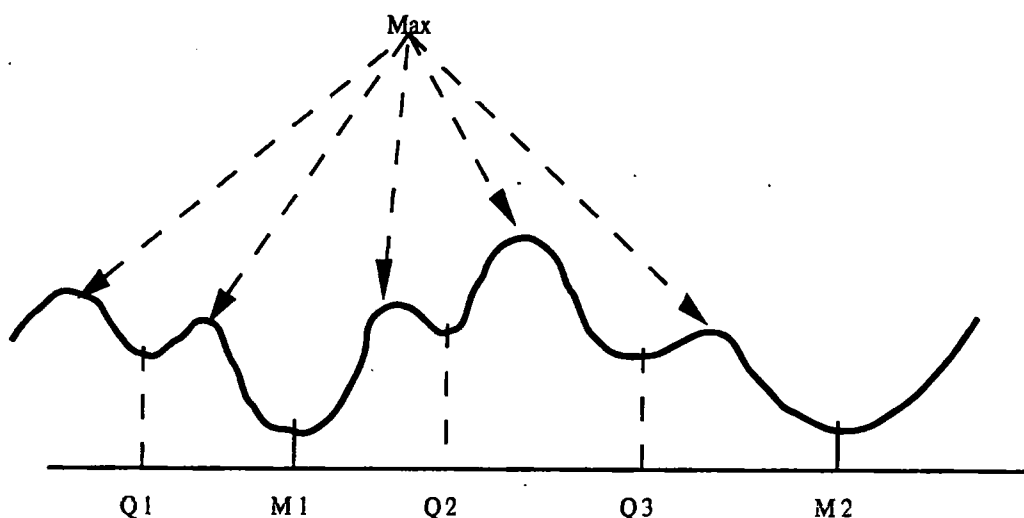
Obviously some of these criteria have different priorities for different researchers. Engineers will be less concerned with the question of biological plausibility than neuroscientists for example. However it is important to realise that there is considerable fertilisation from one discipline to the other. Speech recognition, for example, is of interest to psychologists (How do we recognise speech? can we use neural networks to model some aspect of this process?) and engineers (How do we build a generic speech recognition system?).

One of the most important conceptual frameworks in which neural networks have been examined is to view them as dynamical systems. Broadly speaking, a dynamical system is one which changes over time. In order to view a neural network as a dynamical system, it is necessary to visualise the network as consisting of two sections:

*Representation of network states:* The representation of all the possible states of a dynamical system is known as a state space. Any network state can be viewed as a point on an N dimensional graph, where N is the number of variables that the system has. For a neural network N corresponds to the number of connections (and therefore weights) that the system has.

*"Laws of motion" :* Within the state space: During the learning process the network moves around the phase space, as weight changes as a function of some learning rule take place. The network will of course be moving towards some goal state where the actual output of the network is identical (or within some margin of error) to the desired output. The point on the phase space represents the combination of connection strengths for which, given the appropriate input, transfer and output functions, the desired output will be produced. Such a point is an example of an attractor.

A useful analogy often used in dynamical systems theory to explain this terminology is the landscape metaphor (see figure 1.1). A ball rolls around the landscape, its movement is determined by two factors: the topology of the landscape and the laws of gravity and friction. For a given landscape the ball will only find the global minimum if the energy with which it moves round the landscape is within a certain range. Too little energy will not give it sufficient resources to get out of local minima, whilst too much energy will cause the ball to "escape" from global minimum.



*Figure 1.2 Diagram illustrating the landscape metaphor. Points M1, M2 and Q1 - Q3 are basins of attraction (or attractors) defined by local maxima, labelled in the diagram as Max.*

In figure 1.2 points M1 and M2 are global minima, in a neural network these represent memories or states such that a network in that particular state produces the desired output for a given input. These are global minima within the system. Points Q1 - Q3 represent spurious memories or states. These are local minima within the system, which may cause the network to fail to learn a pattern set, should the network have a set of connection strengths which place the network inside it. Clearly the learning rule should try and stop this from happening by incorporating some

method so that whilst the network might be able to hop out of a local minimum, it would not mistake a global minimum for a local minimum.

One question that might be raised from this comes from the fact that although the patterns which are to be learned during sequence processing tasks are temporal in nature, the state space onto which they are to be mapped is spatial in nature. Thus, how do we get from one to the other? Grinasty et al (1993) reports how temporal order is converted "into spatial correlations expressed in the distributions of neural activities in attractors". Further study of the way in which internal representations are formed during sequential processing tasks was reported by Elman (1991), who stated that distributed representations were formed by the network during learning.

### **1.3 Overview of the Thesis.**

In this section the importance of sequence processing has been made clear, along with a general description of a class of neural network models called recurrent networks. Neural networks have been used by researchers in a wide range of disciplines, nevertheless there are features of a neural network that are desirable to all of them.

Chapter two gives an overview of research into the use of neural networks for sequence processing. Reflecting the current state of research, we concentrate on recurrent network models, describing the major models that have been developed. One problem with this is that researchers often use different terms for the same thing. This is particularly true of mathematical formulae. Accordingly the formulae associated with each network are described as they were by the original authors. Having looked at a wide range of models, efforts that have been made to classify and compare them are then examined.

Chapter three describes a comparative study of three recurrent network architectures over four different sequence processing tasks. This chapter falls into

two parts: i) An examination of the effects of modifying internal network parameters on network performance ii) The comparative study itself.

In chapter four we examine the effects that different modifications to network architectures and learning rules have on the ability of the network to perform sequence processing tasks. In particular we examine modifications to the Real Time Recurrent Learning (RTRL) model of Williams and Zipser (1989).

In chapter five the results of this research are drawn together. We look at the physical properties of the network (number of layers, level of interconnectivity etc.) and see which of these give rise to desirable behaviours (fast learning, ability to generalise etc.).

Finally in chapter six some guidelines for designing more powerful recurrent neural networks are proposed, together with a theoretical metaphor which draws together the findings reported in chapter five. This metaphor looks at the construction of recurrent networks as being more than a matter of changing connection strengths. Finally a number of directions for future research are advocated.

## **1.4 Aims of the thesis**

- To discover what properties of the architecture of a neural network give rise to desirable behavioural properties. These desirable properties are defined as an ability to learn a particular data set, and to generalise from learning to a new set of data from the same problem and to produce the desired output over new data.
- To examine the interaction between the learning rule used by a neural network and its architecture. Can modifications to a learning rule produce different behaviours on the same architecture?
- To see which of these two factors has the greater effect on network performance

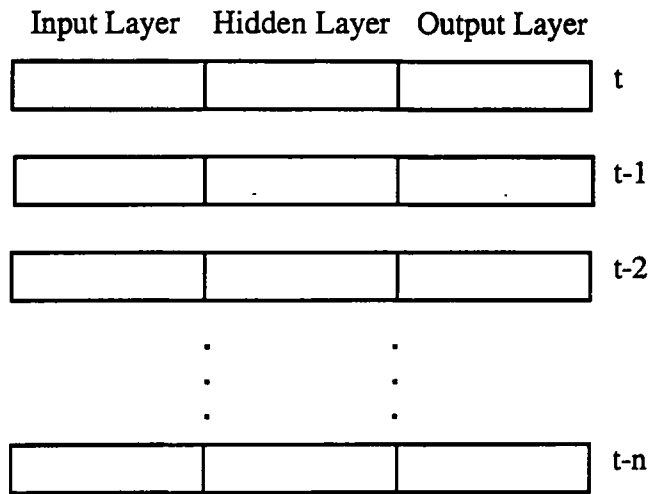
## **Chapter Two. Recurrent Networks: A Review of the Literature**

### **2.1 A history of sequence processing using neural networks.**

A large number of different neural network models have been used in sequence processing problems. They can be divided into one of three classes: i) Feed forward networks ii) Feed forward networks with a shift register attached and iii) Recurrent networks. Of these, recurrent networks have become the dominant type of model used by researchers in sequence processing problems.

#### **2.1.1 Back Propagation Through Time (BPTT)**

The use of feed forward networks for sequence processing problems stems from the observation of Rumelhart, Hinton and Williams (1986) that a multi-layer feed forward network trained with the back propagation learning algorithm is capable of finding a satisfactory solution to almost any problem. In addition Minsky and Pappert (1969) stated that for every recurrent network there is a feed forward network with identical behaviour (over a finite period of time). This is done by adding one layer to the network for each time step needed to represent the sequence. This approach, which is described by Williams and Zipser (1989) as Backpropagation through time (BPTT) has the advantage of great generality and is shown in figure 2.1.

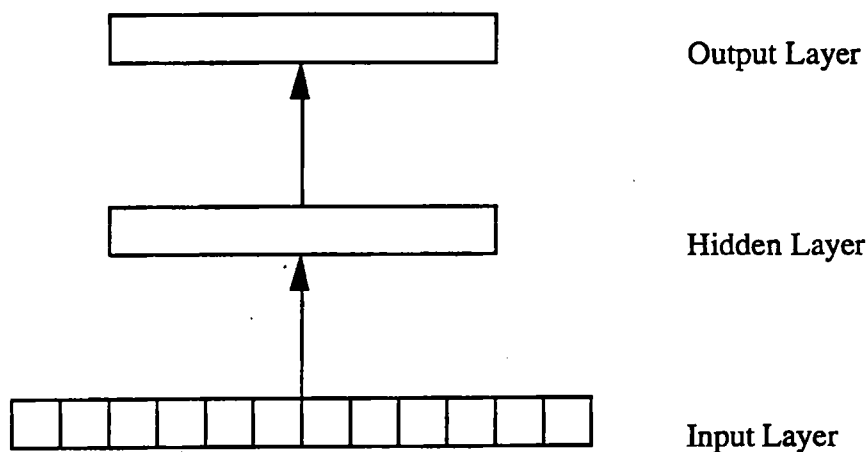


*Fig 2.1. A Schematic Diagram of the Backpropagation through time (BPTT) model, showing how the model grows as the size (n) of the sequence increases.*

However, the difficulty comes when BPTT is implemented in order to solve problems which include long sequences, or where the size of sequence varies widely. In the first case the system requires a great deal of memory, and in the second memory may lie idle for much of the time when the system is processing short sequences.

### **2.1.2 Using Shift Registers for Sequence Processing**

Another approach to tackling sequential processing problems is to add a shift register to a standard feed forward network. The shift register stores information until there is sufficient to perform the task successfully. This is done by presenting an input pattern to the network such that the first element of the sequence is represented by the first portion of the input pattern, the second element of the sequence is represented by the second portion of the input pattern, and so on. A "portion" may represent one or more units in the input layer. All portions are the same size (see figure 2.2).



*Figure 2.2. A feed forward network adapted to perform sequence processing tasks. The input layer is divided into a number of compartments, which contain one or more units. Each compartment represents one element of the sequence. As the network receives each element of the sequence the input layer fills up until it receives the last element of the sequence. At which point the pattern can be spread throughout the network in the traditional manner.*

There are however a number of problems with this approach. As with the BPTT model, the network has to accommodate the largest sequence that it is likely to come across, which may lead to difficulties in tackling problems which include long sequences. Furthermore, as with BPTT a problem where sequence size varies greatly leads to inefficiency. Another difficulty with this approach is that many problems will need to preserve the relative temporal structure of a pattern despite absolute temporal displacement. Consider for example<sup>1</sup> the vectors.

[0 1 1 1 0 0 0 0]

[0 0 0 1 1 1 0 0]

---

<sup>1</sup>This example was taken from Elman (1990).

The dilemma that the network has to resolve is whether the two vectors represent the same structure displaced in time, or as dissimilar structures altogether. They can be taught to recognise them as identical patterns which are temporally displaced, but in doing so the network will have not learned the similarity and the concept will not generalise to novel patterns.

An example of using shift registers is the TRACE model of speech perception described by McClelland and Elman (1986). This model consists of three layers: feature phoneme and word levels. Each unit represents a particular hypothesis concerning the utterance, relative to the start of the utterance. Thus TRACE uses a local as opposed to a distributed representation system. Each bank of feature detectors is replicated over several successive discrete time steps. Input to the model is fed sequentially to the appropriate feature detector at the appropriate time step. The limitations of this type of model that have been discussed above should be obvious: the network becomes computationally expensive if a word needs a large number of time steps to represent it, during which time large parts of the network would be idle. Furthermore a local representation (one unit for each word) does not scale well to large vocabularies.

Because researchers wish to capture the power of the BPTT algorithm whilst avoiding the inefficiencies of shift registers, the majority of neural network models used for sequence processing have been recurrent networks. Recurrent networks are able to represent time by the way in which the previous states of the network (i.e. the previous inputs it received from the outside world) affect the present state of the network. The recurrent connections effectively form a short-term memory mechanism.

### **2.1.3 First and Second Order Recurrent Networks**

Goudreau et al (1994) make a distinction between first order and second order recurrent networks. In a first order network, the parameters are a set of weights  $w_{ij}$



which causes the input (or neuron)  $j$  to have an effect on neuron  $i$ . Thus in a network containing  $M$  inputs and  $N$  neurons, the output of neuron  $i$  at time  $t$  is calculated as follows

$$y_i^t = g\left(\sum_{j=1}^{M+N} w_{ij} z_j^t\right) \quad (2.1)$$

Where  $w_{ij}$  is the connection strength from unit  $j$  to unit  $i$ ,

$$z_j^t = \begin{cases} x_j^t & \text{if } 1 \leq j \leq M \\ y_{j-M}^{t-1} & \text{if } M+1 \leq j \leq M+N \end{cases} \quad (2.2)$$

where  $x_j^t$  is the output of input  $j$  at time  $t$ ,  $y_{j-M}^{t-1}$  is the output of neuron  $j$  at time  $t-1$  and the function  $g$  is a threshold function defined as follows

$$g(x) = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise} \end{cases} \quad (2.3)$$

However a second order recurrent network differs in that there is a set of weights  $w_{ijk}$  which cause neuron  $j$  and input  $k$  to have a combined effect on neuron  $i$ . In such a case the output of a neuron  $i$  is calculated by the following equation

$$y_i^t = g\left(\sum_{j=1}^N \sum_{k=1}^M w_{ijk} y_j^{t-1} x_k^t\right) \quad (2.4)$$

Where the function  $g$  is identical to the threshold function (2.3). Goudreau et al state that a second order recurrent network is able to represent any finite state recogniser, whereas a first order recurrent network cannot.

The difference between first and second order recurrent networks is shown in figure 2.3.

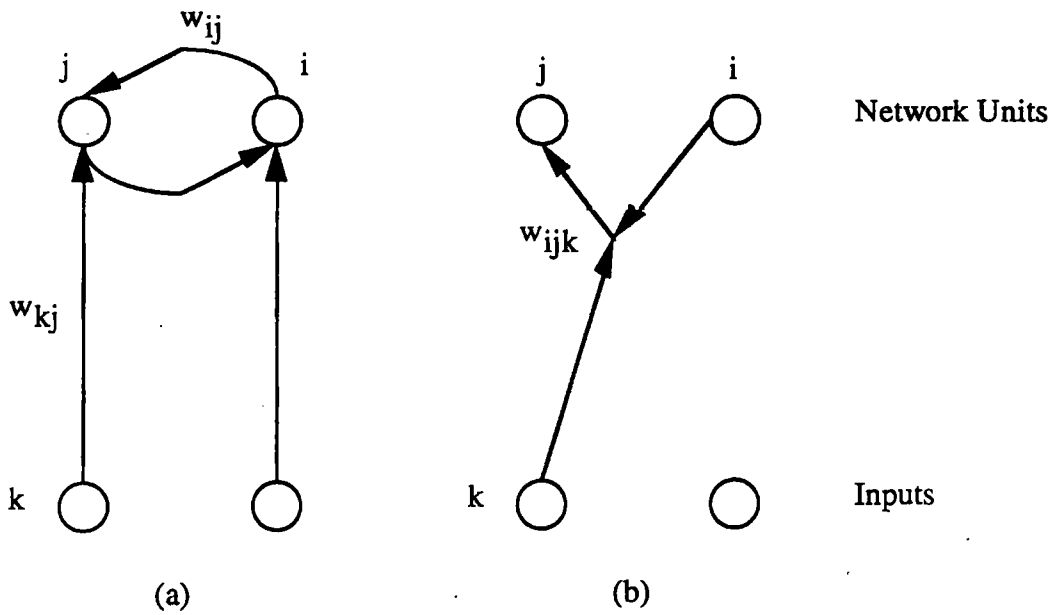


Figure 2.3: (a) A first order and (b) A second order recurrent network. In the first order recurrent network the total input to unit  $j$  is the total input from input lines added to the total input from other units in the network i.e.  $w_{kj} + w_{ij}$ . Whereas in the second order recurrent network input from input lines is multiplied by the input from other units in the network before multiplying by  $w_{ijk}$ . The convention of representing multiplication in this way is taken from Rumelhart and McClelland (1986).

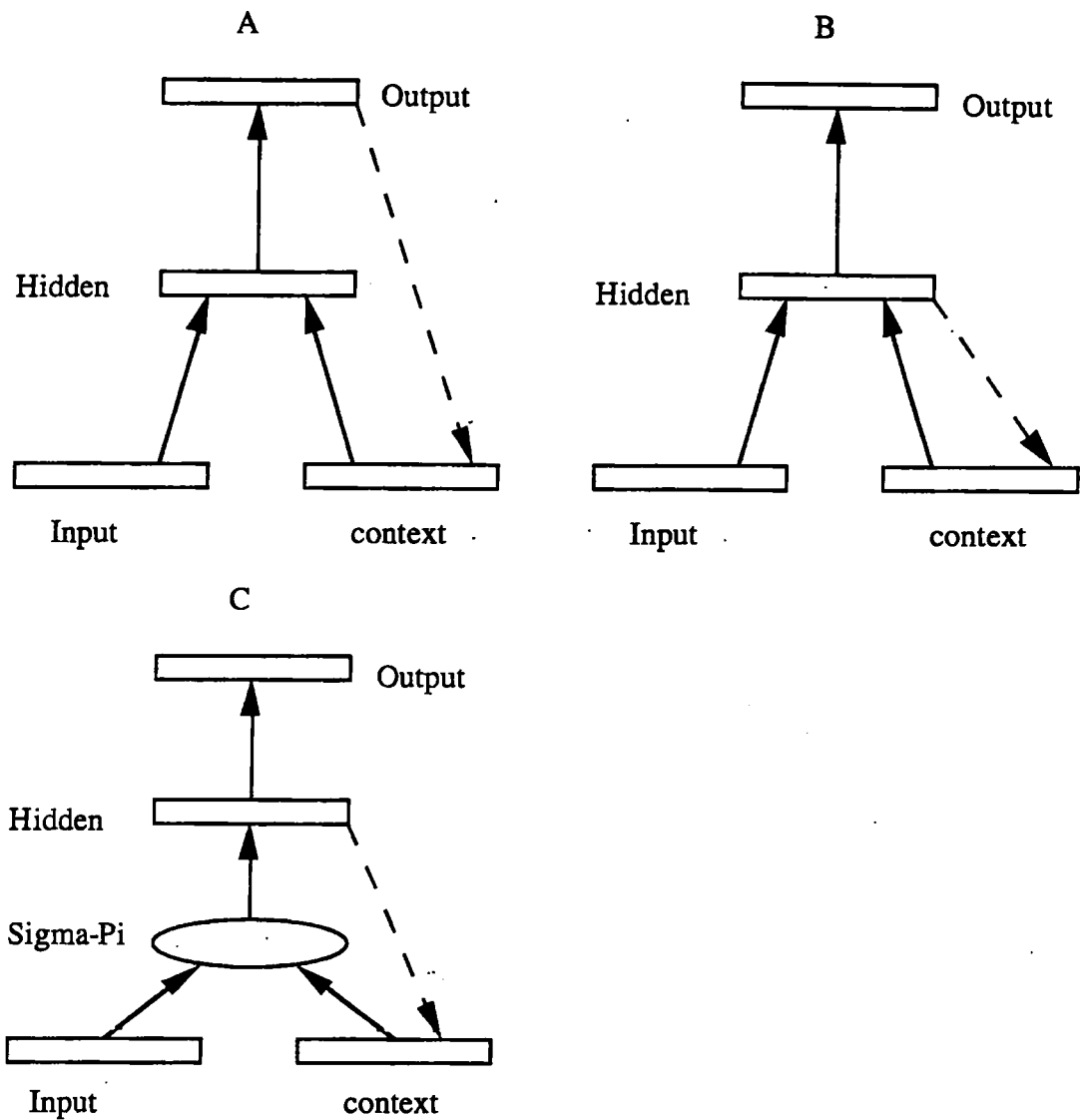
One of the interesting features of recurrent networks when compared to networks with strictly feed forward connections is the richer and more diverse behaviour that recurrent networks display. Dayhoff, Palmadesso and Richards (1994) point out that whereas feed forward networks form fixed point attractors only, recurrent networks can also form periodic oscillations, quasi-periodic oscillations and chaotic attractors. Grinasty, Tsodyks and Amit (1993) describe how the temporal order of a sequence is converted into spatial correlations of the distributions of neural activities in attractors.

#### 2.1.4 The Simple Recurrent Network (SRN)

Recurrent network models often use approximations to the BPTT algorithm. One of the most commonly used models which falls into this category is the Simple Recurrent Network (SRN). The SRN was first proposed by Elman (1990) and has appeared widely throughout the literature since (See for example Cleeremans and McClelland 1991, Servan-Schreiber, Cleeremans and McClelland 1991, Elman 1991, Noda 1994). Elman based the SRN on a model proposed by Jordan (1986) which had a second hidden layer, known as a context layer. Some of the variations on this model are shown in figure 2.4. The short term memory mechanism in the SRN<sup>2</sup> is provided by the feedback to the context layer from the hidden layer. The fixed connections provide the network with a history of its previous hidden layer outputs.

---

<sup>2</sup>In the discussion of the SRN architecture the network discussed will be the one proposed by Elman, shown in figure 2.4B.



*Fig 2.4: Three examples of a Simple Recurrent Network (SRN) architecture. A) As proposed by Jordan (1986) B) As proposed by Elman (1990) C) As proposed by Noda (1994). Layers joined by solid lines are fully interconnected and trainable. Dashed lines indicate 1:1 non trainable connections with a weight value = 1.*

The SRN works as follows for any given time  $t$  for which the network receives information from the "outside world". The input is fed forward through the network and the hidden layer outputs are stored in the context layer. At the next time step  $t+1$ , the hidden layer receives not only input from the outside world but also from its own outputs (stored in the context layer) at time  $t$ . At time  $t+2$  the network

receives the appropriate information from the outside world as well as from itself at time  $t+1$  and so on. Elman set the values of the context units to 0.5 for the first time step, when the network had no history of activations. Weight modification is achieved by using the traditional backpropagation learning algorithm. Thus the network has the power of the backpropagation technique, without the restrictiveness of the BPTT algorithm. Some authors have described the SRN as a truncated BPTT network.

Note that the network proposed by Noda is identical to the one proposed by Elman except for the addition of a layer of sigma-pi units (Rumelhart, Hinton and McClelland 1986). Sigma-pi units differ from more traditional units in that inputs to the unit are multiplied as well as summed. Thus the net input to a standard unit is given by

$$\sum w_{ij} a_i \quad (2.5)$$

where  $w_{ij}$  is the connection strength from unit  $i$  to unit  $j$  and  $a_i$  is the activation of unit  $i$ . Conversely the net input to a sigma-pi unit is given by

$$\sum w_{ij} \prod a_{i1} a_{i2} \dots a_{ik} \quad (2.6)$$

Where  $i$  is an index of the pairs of units, sometimes called conjuncts<sup>3</sup>, which impact on unit  $j$ , and  $a_{i1}, \dots, a_{ik}$  are the  $k$  units in the conjunct. The presence of sigma-pi units in a network make it dynamically programmable in that the output of one unit can directly effect the output of another unit. The most obvious example of this is if

---

<sup>3</sup>Although conjunctions can be of any size, Rumelhart, Hinton and McClelland (1986) state that they have found no application where a conjunct of size  $> 2$  is needed.

one of the units in a given conjunct is zero then the output of the conjunct is zero, no matter how large the output of the other unit in the conjunct.

Servan-Schreiber, Cleeremans and McClelland (1991) argue that an SRN is able to closely mimic a finite state automaton in terms of its behaviour and state representations. They go on to state that it is able to process strings of infinite length even though it has only been trained on strings of finite length. After training an SRN to learn a finite state grammar, Servan-Schreiber, Cleeremans and McClelland (1991) found that not only did activation patterns group according to the different nodes in the finite state grammar, but sub grouping according to the path traversed before reaching the particular node had also taken place.

An alternative method for training SRN models is the TRAINREC algorithm proposed by Kalman and Kwasny (1994). There are three important differences between TRAINREC and the traditional SRN learning algorithm.

*The Error function:* Instead of the usual error function which is some function of the target output minus the actual output, Kalman and Kwasny propose

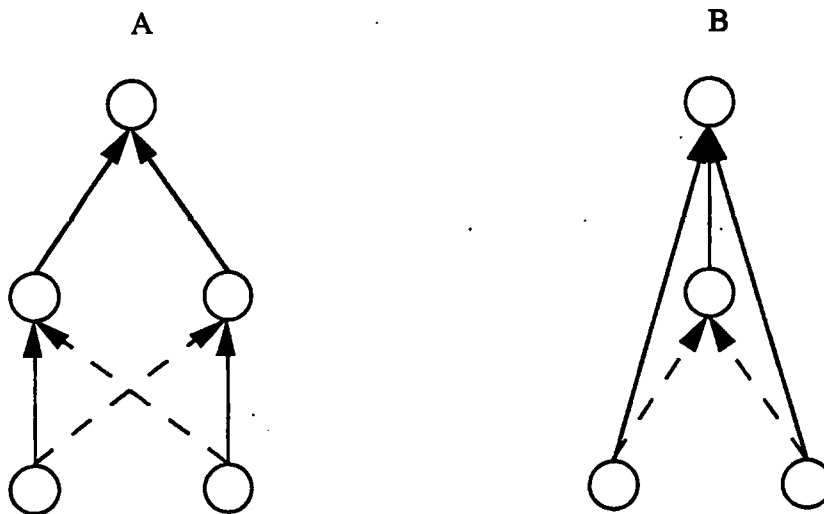
$$error = \frac{(t - a)^2}{1 - a^2} \quad (2.7)$$

Where  $t$  is the target value and  $a$  is the activation of a unit. This error function significantly decreases the number of trials needed for learning to take place (See Table 2.1).

Error Function	Optimisation Method	Epochs	Presentations
Sum of Squares	Backprop	$\infty$	$\infty$
Kalman-Kwasny	Backprop	13000	146900
Sum of Squares	Conjugate Gradient	639	72151
Kalman-Kwasny	Conjugate Gradient	237	26871

*Table 2.1. Table showing differences between error function and optimisation methods of Kalman and Kwasny (1994) over the sum of squares error function and the backpropagation optimisation method. The task measured was training a network to be a deterministic parser (From Kalman and Kwasny 1994 fig 2).*

*Using skip connections:* Skip connections connect units which are not in adjacent layers. The use of skip connections can reduce the number of units needed to perform a particular task. A simple example of this is the XOR problem. Normally this task can be learned with a network of five units minimum. Using skip connections however reduces the minimum number of units to four (see fig 2.5). This in turn reduces the number of degrees of freedom that the network has. The number of variables taken up with skip connections reflects the degree of linearity that the problem has. The higher the number of variables, the more linear the task is.



*Figure 2.5: Minimum size networks to perform the XOR problem (A) without and (B) with skip connections. Dashed lines represent inhibitory connections, solid lines represent excitatory connections.*

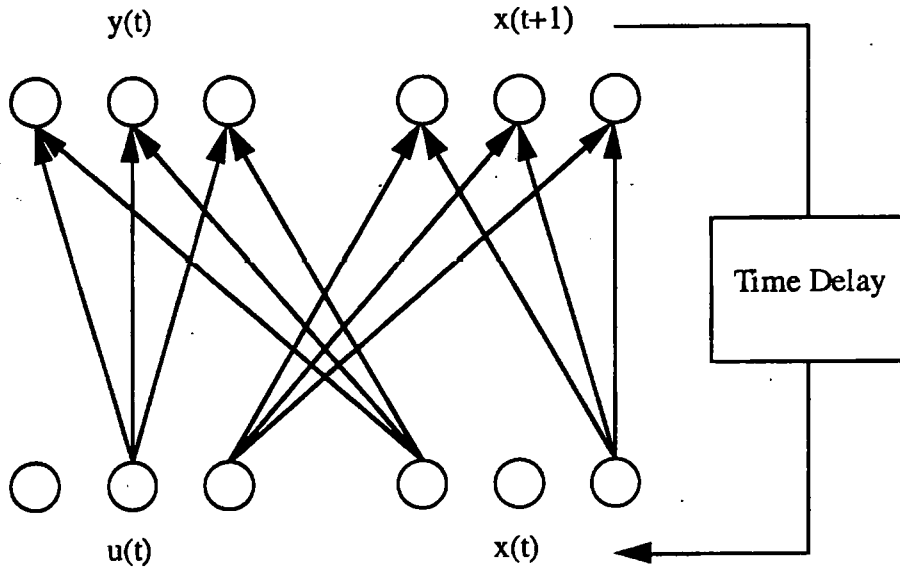
*Singular value decomposition:(SVD):* This is a method of pre-processing information before it arrives at the input layer. SVD allows training to take place when training is not possible on raw data. It may also reduce the number of input units required to represent the sequence. Kalman and Kwasny (1994) use SVD in conjunction with an affine transformation to squash input values into the interval  $[-1,1]$ .

*Correctness criterion:* The usual way to judge the output of a neural network is if the difference between the output and the target output of a network is less than some specified tolerance value. An alternative method proposed by Kalman and Kwasny (1994) is to use Best Vector Match (BVM). This system works best in categorisation tasks where each output unit represents one category. The unit target vector which forms the largest cosine with the unit output vector is the winner. If this category is the same as the target then obviously the answer is correct. Otherwise the weights of the network are modified according to the learning rule used.



The value of the research undertaken by Kalman and Kwasny (1994) is that changing the learning algorithm without changing the architecture of the network can lead to an improvement in the performance of the network.

Robinson, Hochberg and Renals (1995) report on a recurrent network model for speech recognition. The architecture of the model is shown in figure 2.6.



*Fig 2.6: Recurrent network used by Robinson, Hochberg and Renals (1995). In the interests of clarity, not all connections are shown.*

The network receives two inputs: The current input  $u(t)$  and the current state  $x(t)$ . This produces two distinct outputs: The output  $y(t)$  and the next state  $x(t+1)$ . If we take the combined input to be  $z(t)$  and the weight matrices to the outputs and the next state as  $W$  and  $V$  respectively, then:

$$z(t) = \begin{bmatrix} 1 \\ u(t) \\ x(t) \end{bmatrix} \quad (2.8)$$

Where 1 is included to apply a bias mechanism.

$$y_i(t) = \frac{\exp(W_i z(t))}{\sum_j \exp(W_j z(t))} \quad (2.9)$$

$$x_i(t+1) = \frac{1}{1 + \exp(-V_i z(t))} \quad (2.10)$$

Robinson, Hochberg and Renals (1995) use backpropagation through time as the algorithm for calculating the value of the weight change during learning. The state units are treated as hidden units in a traditional feed forward network since they have no target values attached to them. The value of the weight change  $\frac{\delta E^{(n)}}{\delta W_{ij}^{(n)}}$  is then used to update the weights according to the following formula:

$$w_{ij}^{(n+1)} = \begin{cases} w_{ij}^{(n)} + \Delta w_{ij}^{(n)} & \text{if } \frac{\delta E^{(n)}}{\delta W_{ij}^{(n)}} < 0 \\ w_{ij}^{(n)} - \Delta w_{ij}^{(n)} & \text{otherwise} \end{cases} \quad (2.11)$$

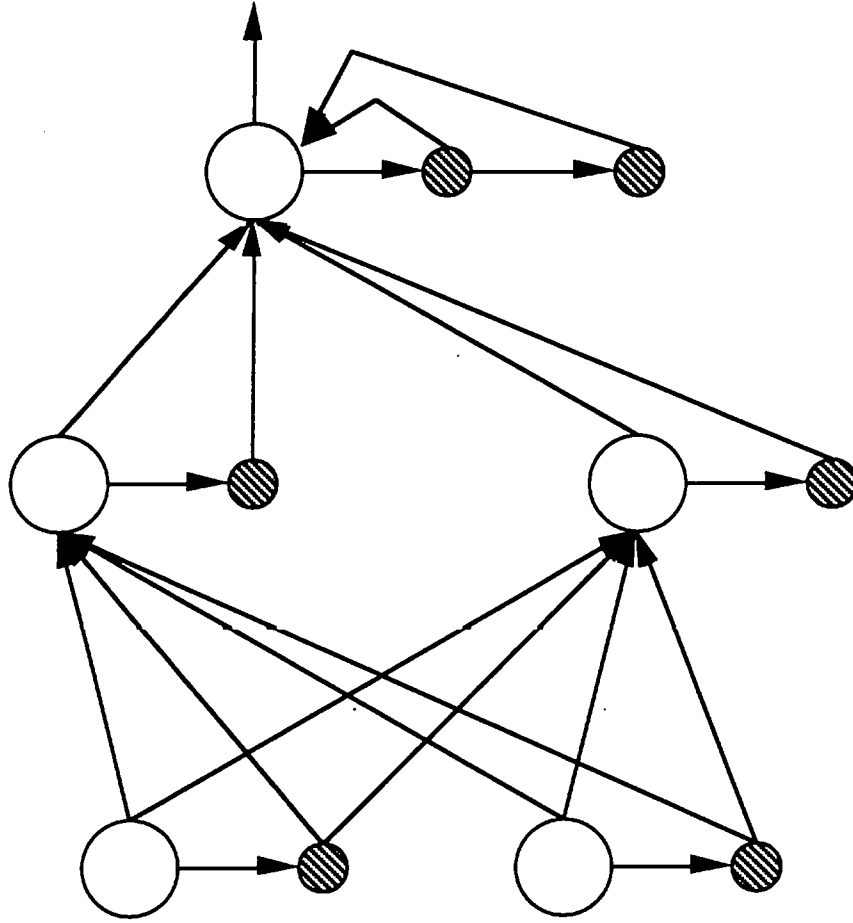
This model also uses a pruning algorithm to remove unnecessary connections and therefore increase the speed of the network. It was used as part of a system where more than one network was 'combined'. This was done by averaging the outputs of four such networks, although more sophisticated averaging methods are available. Each network differed in the way in which pre-processing of data was performed. It was found that this led to a 17% reduction of error. The performance of the model over a range of speech recognition data sets is shown in table 2.2. The terms MEL+ and PLP refer to forms of standard acoustic vector representations. The former is "a twenty channel power normalised mel-scaled filter bank representation augmented with power, pitch and degree of voicing" (Robinson, Hochberg and Renals 1995 pp7). The latter is "a twelfth order perceptual linear prediction cepstral coefficients plus energy" (Robinson, Hochberg and Renals 1995 pp7).

Merge Type	Word Error Rate		
	Spoke 5	Spoke 6	H2
Forward MEL+	17.3	15.0	16.2
Forward PLP	17.1	15.1	16.5
Backward MEL+	17.8	15.5	16.1
Backward PLP	16.9	14.4	15.2
Average	17.3	15.0	16.0
Uniform Merge	15.2	11.4	13.4
Log-Domain	13.4	11.0	12.6

*Table 2.2 Performance of the recurrent network model of Robinson, Hochberg and Renals (1995). Figures shown are percentage error.*

### 2.1.5 Memory Neural Networks

The use of recurrent networks of the type found in the SRN was taken a step further with the memory neural network (Surkan and Skurikhin 1994). There are recurrent connections at all three layers of the network. Each network neuron has an associated memory neuron, which as the name suggests holds information concerning the activity of the network neuron from previous time steps. The architecture of a memory neural network is shown in figure 2.7.



*Fig 2.7. A typical memory neural network architecture. Clear circles represent network neurons, shaded circles represent memory neurons. In the interests of clarity, not all neurons and connections are shown.*

As can be seen in the diagram, a network neuron in a given layer  $l$  receives input from both the memory and network neurons in the layer below it. All neurons in layer  $l$  feed forward to only the network neurons in layer  $l+1$ . A given memory neuron will only receive input from its associated network neuron. The input to the  $j$ th network neuron of layer  $l$  at time  $t$  is calculated as follows:

$$x_j^l(t) = \sum_{i=0}^{N_{l-1}} w_{ij}^{l-1} \cdot u_i^{l-1}(t) + \sum_{i=1}^{N_{l-1}} f_{ij}^{l-1} \cdot v_i^{l-1}(t) \quad (2.12)$$

For the network neurons at the output layer, however, the total input is as follows:

$$x_j^L(t) = \sum_{i=0}^{N_{l-1}} w_{ij}^{L-1} \cdot u_i^{L-1}(t) + \sum_{i=1}^{N_{l-1}} f_{ij}^{L-1} \cdot v_i^{L-1}(t) + \sum_{i=1}^{M_j} \beta_{ij}^L \cdot v_{ij}^L(t) \quad (2.13)$$

Where  $N_l$  is the number of network neurons in the  $l$ -th layer.  $M_j$  is the number of memory neurons associated with the  $j$ -th network neuron of the output layer.  $u_j^i(t)$  is the output of the network neuron at time  $t$ .  $v_j^i(t)$  is the output of the corresponding memory neurons at time  $t$ .  $\beta_{ij}^l(t)$  is the connection strength from the  $i$ -th memory neuron of the  $j$ -th network neuron to the  $j$ -th network neuron in the output layer.  $w_{ij}^l$  is the connection strength from network neuron  $i$  to network neuron  $j$  (where  $j$  is a unit in layer  $l+1$ ).  $f_{ij}^l$  is the connection strength from the corresponding memory neuron of neuron  $i$  to the  $j$ -th network neuron.

To produce an output, the total input is passed through a transfer function. The output for all memory neurons not in the output layer is calculated thus:

$$v_j^i(t) = \alpha_j^i \cdot u_j^i(t-1) + (1 - \alpha_j^i) \cdot v_j^i(t-1) \quad (2.14)$$

for the memory neurons in the output layer the output is calculated according to the following formula:

$$v_{ij}^L(t) = \alpha_{ij}^L \cdot v_{ij-1}^L(t-1) + (1 - \alpha_{ij}^L) \cdot v_{ij}^L(t-1) \quad (2.15)$$

Where  $v_{i0}^l = u_i^l$ . The network will remain stable provided that  $0 \leq \alpha_{ij}^l, \alpha_i^l, \beta_{ij}^l \leq 1$ .

During learning there is modification of connection strengths between network neurons, between memory and network neurons as well as modification of the memory coefficient for memory neurons. The learning rule employed is a modification of the traditional back propagation algorithm.

Surkan and Skurikhin tested their network on its ability to predict daily energy usage in a large geographical area. They found that the learning process converged in fewer than 5000 iterations, with the majority of learning being done in

the first 100 to 2500 iterations of learning. The networks were tested on their ability to predict energy usage for a range of times into the future (one, two or three days ahead). It was found that accuracy decreased as the network was asked to predict further into the future.

Memory neural networks were also used by Fallon Garcia and Tummala (1994) as a missile guidance system. They found that whilst the network was good at predicting target velocity, it did rather less well at predicting target position.

### 2.1.6 Finite Impulse Response (FIR) filters

An alternative modification of a traditional feed forward network is to use Finite Impulse Response (FIR) filters. An example of this method is described by Wan (1993). The network operates by passing input signals to a unit through FIR filters. The filter works as follows: Past samples of the input can be represented as a vector  $\mathbf{x}(t) = [x(t), x(t-1), \dots, x(t-N)]$ . Also there is a weight vector for the filter coefficients  $\mathbf{w} = [w(0), w(1), \dots, w(N)]$ . The static weight of the original feed forward network can now be replaced by the FIR filter. The values at time  $t$  of the activation and output ( $y(t)$  and  $out(t)$  respectively)<sup>4</sup> for each unit in the network is now defined as follows

$$y(t) = \sum_i \mathbf{w}_i \cdot \mathbf{x}_i(t) \quad (2.16)$$

$$out(t) = f[y(t)] \quad (2.17)$$

The network learns by using a rule called temporal backpropagation. The formula for which is as follows

---

<sup>4</sup> In Wan (1993) from where this description of FIR filters is taken time is denoted as  $k$ , which has been changed in this description to the more standard  $t$ .

$$w_{ij}^l(t+1) = w_{ij}^l(t) - \mu \delta_j^{l+1}(t) \cdot x_i^l(t) \quad (2.18)$$

$$\delta_j^l(t) = \begin{cases} -2e_j(t)f'(y_j^l(t)) & l = L \\ f'(y_j^l(t)) \cdot \sum_{m=1}^{N+1} \delta_m^{l+1}(t) \cdot w_{jm}^l & 1 \leq l \leq L-1 \end{cases} \quad (2.19)$$

Where  $e_j(t)$  is the error of unit  $j$  at time  $t$ ,  $w_{jm}^l$  specifies the coefficients for the connecting filter  $y_{ij}^l$  is a filter connecting unit  $i$  in layer  $l-1$  to unit  $j$  in layer  $l$  and  $L$  is the number of layers in the network. The learning process is applied layer by layer working from the output layer to the input layer.

Wan (1993) demonstrates the ability of a neural network using a FIR filter structure. The model was tested on a chaotic time series. In this case the intensity pulsation of an NH3 laser. For the training data Wan used 1000 samples from the data set, the task of the network being to predict the next 100 samples. The neural network outperformed all the other methods used, which included standard recurrent and feed forward networks.

Another model to utilise a filter structure is the Infinite Impulse Response (IIR) synapse multi-layer perceptron advocated by Back and Tsoi (1991). The synapses of this model have a linear transfer function (see fig 2.8) The neuron used by Back and Tsoi (1991) is a modification of the McCulloch-Pitt neuron

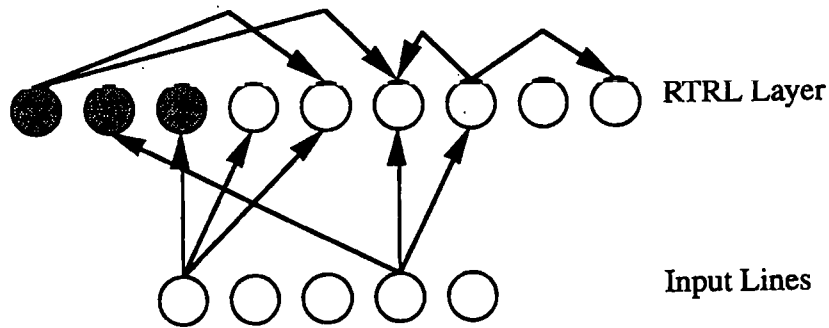
$$y(t) = f\left(\sum_{i=0}^n G_i(z^{-1})x_i(t)\right) \quad (2.20)$$

Where  $x_i(t)$  is the input to the neuron from the previous layer and  $G_i(z^{-1})$  is a linear transfer function. The inputs to the neuron may be taken from either the previous layer (in which case it is local synapse feedback) or from the output of the unit (in which case it is local output feedback). In a comparative study (see sections 2.2 and 2.3), Tsoi and Back argue that different positions of feedback give rise to different kinds of behaviour. A model referred to by Back and Tsoi (1991) as the Frasconi-

Gori-Soda architecture differs from the model designed by Back and Tsoi (1991) in that the feedback occurs after the non linearity introduced to the system by the transfer function (see fig 2.8).

### 2.1.7 The Real Time Recurrent Learning model, and its variations.

Whilst acknowledging the limitations of the BPTT approach, other researchers have tried to capture its generality in a more practical framework. The Real Time Recurrent Learning (RTRL) model of Williams and Zipser (1989) is an example of this. The general RTRL model has an unrestricted architecture i.e. every unit in the network is connected to all other units in the network. Information from the outside world is fed to the system by external input lines, each of which connects to all the units in the RTRL network. An example of an RTRL network is shown in figure 2.8.



*Figure 2.8 A Real Time Recurrent Learning (RTRL) network. The subset of output units are shaded. Not all connections between units are shown.*

During learning, the connection  $W_{ij}$  is updated according to the following equation:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \sum_{k \neq i} e_k(t) p_{ij}^k(t) \quad (2.21)$$

where  $\alpha$  is a constant called the learning rate, and



Where  $e_k(t)$  is an error term such that:

$$e_k(t) = \begin{cases} d_k(t) - y_k(t) & \text{If } k \in T(t) \\ 0 & \text{Otherwise} \end{cases} \quad (2.22)$$

Where  $T$  is the subset of RTRL units designated as output (or target) units,  $d_k(t)$  is the desired output of unit  $k$  and  $y_k(t)$  is the actual output of unit  $k$ . The second part of the learning rule  $p_{ij}^k$  is a dynamic variable called impact which measures the sensitivity of the output value of unit  $k$  at time  $t$  to a small change in the weight  $w_{ij}$ . This sensitivity is calculated as follows:

$$p_{ij}^k(t+1) = f'_k[s_k(t)] \left[ \sum_{l \in u} w_{kl} p_{ij}^l(t) + \delta_{ik} z_j(t) \right] \quad (2.23)$$

with initial conditions:

$$p_{ij}^k(t_0) = 0 \quad (2.24)$$

Where:  $z_j(t)$  is the output of neuron  $j$ , and

$$f'_k[s_k(t)] = y_k(t+1)[1-y_k(t+1)] \quad (2.25)$$

and  $\delta_{ik}$  denotes the Kronecker delta:

$$\delta_{ik} = \begin{cases} 1 & \text{if } i = k \\ 0 & \text{Otherwise} \end{cases} \quad (2.26)$$

Note that equation 2.25 is only applicable when a sigmoid transfer function is used. Perhaps the most obvious feature of this learning rule is that, because of the impacts, changing one weight is in part a function of all the other units in the network. Thus the RTRL learning rule is non-local. However no replication of the

network over time is required. The amount of storage and computation depends not on the size of the sequence to be processed but instead on the size of the network. Williams and Zipser (1989) show that for a network with  $n$  units and  $m$  external input lines there are  $n^3 + nm^2$   $p_{ij}^k$  values.

Williams and Zipser (1989) tested their network on a range of tasks, using both the original and the teacher forcing RTRL algorithm. There were tasks that both versions of RTRL were capable of performing. An example of which is the XOR task modified so that a delay is introduced. The difference between this and the standard version of the XOR task is that not only does the network have to be capable of learning the task, but it also has to hold inputs from individual time steps in its memory mechanism. However other tasks did reveal differences in performance between the two versions of RTRL. An example of which is when Williams and Zipser (1989) taught the network to produce oscillatory outputs: training a single unit to produce the sequence 010101... or a two unit set to produce 00110011... then only the teacher forcing version of RTRL is able to perform this task.

One variant of the learning rule described above is to use a technique described by Williams and Zipser as teacher forcing. In this method, the actual output of the network is replaced by a teacher signal (i.e. the desired output). Under the original RTRL algorithm the outputs of the network could be denoted thus

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in I \\ y_k(t) & \text{if } k \in U \end{cases} \quad (2.27)$$

Whereas under RTRL with teacher forcing the outputs of the network are as follows

$$z_k(t) = \begin{cases} x_k(t) & \text{if } k \in I \\ d_k(t) & \text{if } k \in T(t) \\ y_k(t) & \text{if } k \in U - T(t) \end{cases} \quad (2.28)$$

In equations (2.27) and (2.28)  $I$  denotes the input lines to an RTRL network,  $U$  denotes the units in the RTRL network itself.  $T$  denotes the subset of RTRL units that are target units. Another feature of RTRL with teacher forcing is that the  $p_{ij}^k$  values are set to zero for all target units after each weight update. Thus the learning rule is similar to equation (2.23).

$$p_{ij}^k(t+1) = f'_k(s_k(t)) \left[ \sum_{l \in U-T(t)} w_{kl} p_{ij}^l(t) + \partial_{ik} z_j(t) \right] \quad (2.29)$$

Several modifications to the RTRL model have been suggested so as to improve performance. Schmidhuber (1992) suggests an algorithm which combines the RTRL algorithm with the BPTT algorithm. The training data is broken down into blocks, a block consists of a number of discrete time steps. If we use the error calculation as defined in equation (2.22), the total error over all target units  $k$  is calculated as follows:

$$E(t) = \frac{1}{2} \sum_{k \in U} [e_k(t)]^2 \quad (2.30)$$

Whilst the total error over the time interval  $(t, t')$  i.e. one block is defined as:

$$E^{total}(t', t) = \sum_{\tau=t'+1}^t E(\tau) \quad (2.31)$$

Schmidhuber uses the following notation to describe his improved learning rule:

$U$  is the set of indices  $k$  such that at time  $t$ ,  $x_k(t)$  is the output of non input unit  $k$ .

$I$  is the set of indices  $k$  such that at time  $t$ ,  $x_k(t)$  is the external input of input unit  $k$ .

$T(t)$  denotes the units for which there is a target value  $d_k(t)$  at time  $t$ .

$w_{ij}$  denotes the connection strength from unit  $j$  to unit  $i$ .

The algorithm can be divided into the following steps:

i) Compute the contribution of  $E^{total}(0, t_0+h)$  to the change in  $w_{ij}$ :

$$\Delta w_{ij}(t_0 + h) = -\alpha \sum_{\tau=1}^{t_0+h} \frac{\partial E^{total}(0, t_0 + h)}{\partial w_{ij}(\tau)} \quad (2.32)$$

Where  $\alpha$  is a constant.

ii) From time interval  $t_0$  to  $t_0+h$  let the network run according to the dynamics described as follows:

$$x_k(t) = f_k[net_k(t)] \quad (2.33)$$

Where  $f_k$  is a differentiable (usually semi-linear) function and

$$net_k(t+1) = \sum_{l \in U \cup I} w_{kl}(t+1)x_l(t), \quad net_k(0) = 0 \quad (2.34)$$

iii) Perform calculations for calculating error derivatives similar to those used in the standard Real Time Recurrent Learning algorithm of Williams and Zipser (1989), but in a manner similar to the Back Propagation Through Time algorithm:

$$\begin{aligned} \frac{\partial E^{total}(0, t_0 + h)}{\partial w_{ij}} &= \frac{\partial E^{total}(0, t_0)}{\partial w_{ij}} + \sum_{\tau=1}^{t_0} \frac{\partial E^{total}(0, t_0 + h)}{\partial w_{ij}(\tau)} \\ &+ \sum_{\tau=t_0+1}^{t_0+h} \frac{\partial E^{total}(0, t_0 + h)}{\partial w_{ij}(\tau)} \text{ where } \frac{\partial E^{total}(0, 0)}{\partial w_{ij}} = 0 \end{aligned} \quad (2.35)$$

Where

$$\sum_{\tau=t_0+1}^{t_0+h} \frac{\partial E^{total}(0, t_0 + h)}{\partial w_{ij}(\tau)} = - \sum_{\tau=t_0+1}^{t_0+h} \delta_i(\tau) x_j(\tau-1) \quad (2.36)$$

$$\delta_i(\tau) = \begin{cases} f'_i[net_i(\tau)] e_i(\tau) & \text{if } \tau = t_0 + h \\ f'_i[net_i(\tau)] \left[ e_i(\tau) + \sum_{l \in U} w_{li} \delta_l(\tau+1) \right] & \text{if } t_0 \leq \tau \leq t_0 + h \end{cases} \quad (2.37)$$

$$\sum_{\tau=1}^{t_0} \frac{\partial E^{total}(0, t_0 + h)}{\partial w_{ij}(\tau)} = - \sum_{\tau=t_0+1}^{t_0+h} \delta_k(t_0) q_{ij}^k(t_0) \quad (2.38)$$

iv) Compute  $q_{ij}^l(t_0 + h)$  for all possible  $l, i$  and  $j$ . Perform one set of calculations for each  $l$  according to the following formulae:

$$q_{ij}^l(t_0 + h) = \sum_{k \in U} \gamma_{ik}(t_0) h_{ij}^k(t_0) + \sum_{\tau=t_0+1}^{t_0+h} \gamma_{il}(\tau) x_j(\tau-1) \quad (2.39)$$

$$q_{ij}^l(0) = 0$$

$$\text{if } \tau = t_0 + h: \gamma_{li}(\tau) = \begin{cases} 1 & \text{if } l = i \\ 0 & \text{otherwise} \end{cases} \quad (2.40)$$

$$\text{else if } t_0 \leq \tau \leq t_0 + h: \gamma_{li}(\tau) = f'_i[net_i(\tau)] \sum_{a \in U} w_{ai} \gamma_{la}(\tau+1) \quad (2.41)$$

v) Set  $t_0 = t_0 + h$  and go back to step i.

The main advantage of this algorithm according to Schmidhuber is that the average number of calculations per time step is  $O(n^3)$  compared to  $O(n^4)$  in the original RTRL algorithm. Schmidhuber tested this modification on what was termed a "flip flop" task. This is defined as follows: the desired output of the network is 1 as soon as an event B follows an event A and 0 at all other times. The key aspect of this problem is that unlike the XOR with delay task, the length of interval between events A and B is unknown and indeed may vary widely, thus a more powerful and flexible memory representation is required. Williams and Zipser found that both the original and teacher forced RTRL algorithms were typically able to solve this problem after

5000 presentations. The modified algorithm could solve the problem after 300 presentations.

Another method for improving the performance of the RTRL model was suggested by Zipser (1989). This involves dividing the network into fully recurrent sub networks. It is assumed that the sub networks are of equal size. The only criteria for dividing the network is that each of the sub networks must have at least one of the "target" units as a member. Simply dividing a network into two sub networks will lead to the  $p_{ij}^k$  values being calculated four times more quickly than in an undivided network.

Zipser found that his modified RTRL model performed identically to the original RTRL algorithm, indeed the connection strengths for the two were often quite similar. Differences appeared however when the models were compared on a "Turing machine" test, where the network sees the inputs and outputs of a finite state machine, but not the internal structure of the machine, which the network must create. Whereas the original RTRL algorithm was able to successfully perform the task on 50% of attempts, Zipser's subdivided network could only do so when teacher forcing was used, again the subdivided network was only successful 50% of the time.

RTRL, however, is not the only algorithm developed for training fully connected networks. Metzger and Lehmann (1994) describe a learning rule where the units are divided into two groups defined by the nature of their outputs. Units can be either excitatory or inhibitory. At a given time step, units are chosen at random and their state is updated. The two main parts of the updating equation are i) the input to neuron i:

$$h_i = h(i, S) + H_i - U_i \quad (2.42)$$

and ii) the noise level  $T$  of the network. In equation (2.42)  $h(i, S)$  is the input that  $i$  receives from other neurons in the network,  $H_i$  is the input that the network receives

from the outside world and  $U_i$  is a threshold. Updating is performed according to the following rule:

$$S_i = \begin{cases} 1 & \text{With probability } f(h_i) \\ 0 & \text{With probability } 1 - f(h_i) \end{cases} \quad (2.43)$$

where

$$f(h_i) = [1 + \exp(-h_i / T)]^{-1} \quad (2.44)$$

The units are divided equally into the two sets of excitatory and inhibitory units. The connections linked to inhibitory units are constant and are not modified during learning. Because there are no connections between inhibitory units their only function is to regulate the activity of the excitatory units. Weight updates are performed according to a hebbian learning rule.

The mechanism developed by Williams and Zipser (1989) for training an RTRL network has also been used by Schmidhuber (1992) to show an alternative to recurrent networks for sequence processing problems. Instead the model consists of two feed forward networks. The first net learns to produce the weight changes for the second net (called a fast weight network). Schmidhuber claims that because this model does not require a fully fledged feedback system to provide a memory mechanism (indeed in some cases a single weight may be sufficient) this provides an opportunity for increased storage capacity. If the network which outputs the weight changes has one output unit per weight then its weight update is as follows

$$\Delta w_{ij} = -\eta \sum_{w_{ab} \in W_F} \delta_{ab}(t) p_{ij}^{ab}(t-1) \quad (2.45)$$

Where  $\eta$  is a constant learning rate and

$$p_{ij}^{ab}(t) = \frac{\partial \sigma[w_{ab}(t-1), s_{ab}(t)]}{\partial w_{ab}(t-1)} p_{ij}^{ab}(t-1) + \frac{\partial \sigma[w_{ab}(t-1), s_{ab}(t)]}{\partial s_{ab}(t)} \frac{\partial s_{ab}(t)}{\partial w_{ij}} \quad (2.46)$$

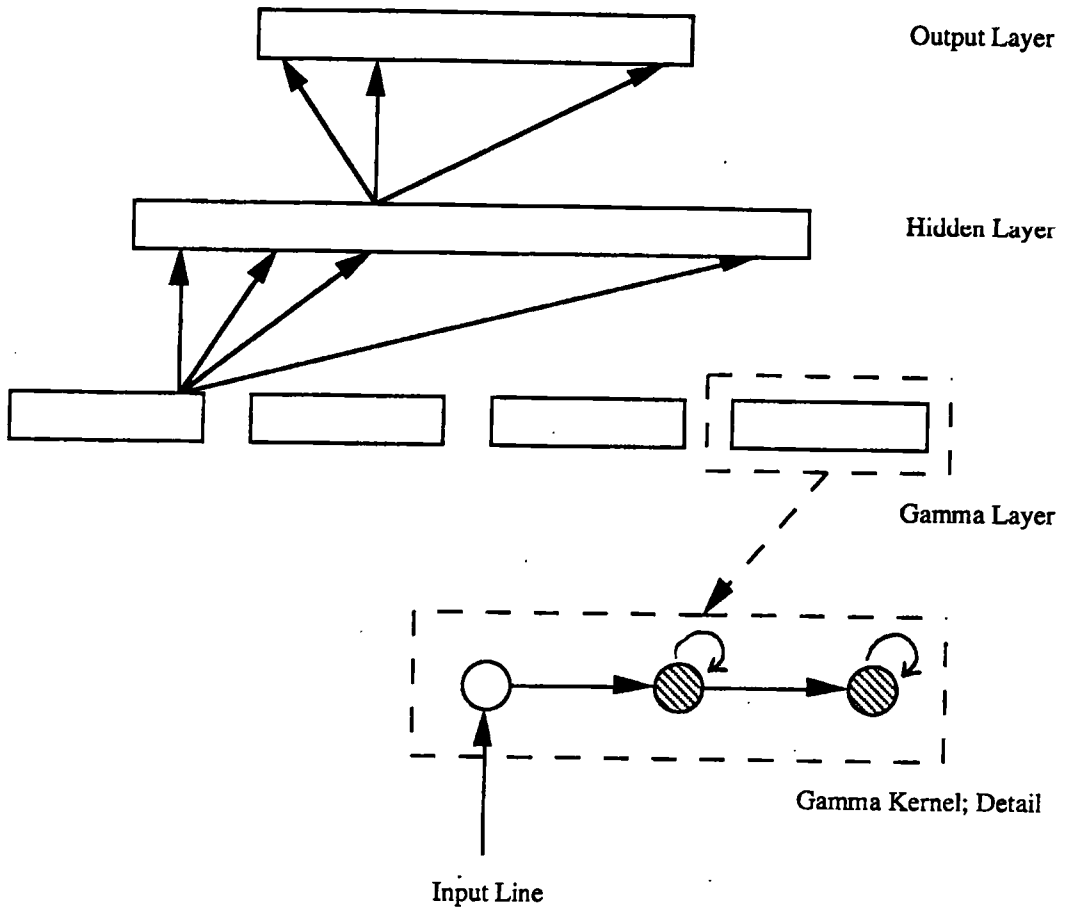
Where  $w_{ab}$  is a weighted connection from unit  $a$  to unit  $b$  and  $s_{ab}(t)$  is the output units activation at time  $t$ . However in the above equation the number of output units grows in proportion to the size of the network. To combat this, Schmidhuber proposes that the feed forward network should have an output unit for each unit from which a fast weight originates and an output unit for each unit to which a fast weight leads. In this case the term  $s_{ab}(t)$  is replaced by  $s_a(t)s_b(t)$ .

### 2.1.8 The Gamma Model

In the recurrent network models discussed above, note that the memory mechanism used by the network is static. In such a situation the rest of the network has to modify its connections so that a task can be successfully performed. However, it should be clear that not all sequence processing problems require the same type of memory mechanism, both in terms of the number of items that need to be stored in memory and the form that these items should take. There has also been much research into recurrent network models which have dynamic memory mechanisms, which allow the network to form memory structures appropriate to a particular task.

An example of a recurrent network with a dynamic memory structure is the gamma model (de Vries and Principe 1992; Principe, de Vries and de Oliveira 1993). In this case it is more appropriate to describe the gamma model as a gamma memory structure, since de Vries and Principe claim that it can be 'bolted on' to a number of different feed forward network models, turning them into recurrent networks capable of dealing with sequence processing problems. An example of a gamma network is shown in figure 2.9.





*Figure 2.9. A gamma network. In this case the gamma memory structure connects to a feed forward network. Shaded units in the gamma kernels have feedback connections to themselves.*

The gamma memory structure consists of groups of units, called kernels. Each kernel has a dedicated input line, whose output at any time is equal to the input it receives from the outside world. All units in the gamma memory structure feed forward to the next layer in the network. The output of units in a gamma kernel are calculated as follows:

$$x_i(t) = I_i(t) \quad (2.47)$$

$$x_{ik}(t) = (1 - \mu)x_{ik}(t-1) + \mu x_{i,k-1}(t-1) \quad (2.48)$$

Equation (2.47) relates to the input line to the kernel. Equation (2.48) relates to members of the kernel. The key to the way in which the gamma memory structure works is the variable  $\mu$ . The memory is a trade-off between two factors: memory depth and resolution. Memory depth refers to how many time steps back information is held, whereas resolution refers to the degree of detail in which the information is kept. A kernel with high memory depth will have a coarse grained description of its previous states and vice versa.

This principle can be seen in equation 2.48. The unit  $x_{ik}$  at time  $t$  receives information from the previous unit in the gamma kernel and from itself from the previous time step. The former is multiplied by  $\mu$ , the latter by  $1-\mu$ . Thus the higher the value of  $\mu$  the more information from the previous unit in the gamma kernel will be preserved and more information concerning the output  $x_{ik}$  at  $t-1$  will be lost. The opposite will be the case as the value of  $\mu$  decreases.

During learning the value of  $\mu$  will be updated according to the following equations:

$$\Delta\mu = -\sum_m e_m(t) \sigma'_m(net_m(t)) \sum_k w_{mik} \alpha_i^k(t) \quad (2.49)$$

Where

$$\alpha_i^k(t) = (1 - \mu_i) \alpha_i^k(t-1) + \mu_i \alpha_i^{k-1}(t-1) + [x_{i,k-1}(t-1) - x_{i,k}(t-1)] \quad (2.50)$$

In these equations  $e_m(t)$  is desired output - actual output,  $\sigma'_m(net_m(t))$  is the back propagated error from the layer(s) above the gamma memory structure and  $w_{mik}$  is the connection strength between units  $i$  and  $k$ .

A variation on this model was proposed by Principe and Turner (1994) whose gamma memory structure differed in that the memory parameter  $\mu$  was adapted locally. Hence there will be a different value of  $\mu$  for every unit in the gamma

kernel. This leads to a gamma network having a composite memory depth and resolution. The memory depth  $\mathfrak{S}_{\mu 0k}$  is given by:

$$\mathfrak{S}_{\mu 0k} = C^{k-1} \sum_{i=1}^k \mathfrak{S}_{\mu i-1i} \quad (2.51)$$

Where

$$\mathfrak{S}_{\mu i-1i} = C\mu_i^{-1} \quad i \rightarrow [1, N_\mu] \quad (2.52)$$

Composite memory resolution  $\mathfrak{R}_{\mu 0N_\mu}$  is calculated according to the following formulae:

$$\mathfrak{R}_{\mu 0N_\mu} = N_\mu C^{-N_\mu+1} \left( \sum_{i=1}^{N_\mu} \mathfrak{S}_{\mu i-1i} \right)^{-1} \quad (2.53)$$

As with the original implementation of the gamma model, the memory structure mapped onto a multi-layer perceptron.

## 2.2 Issues of Convergence

Research into the use of neural networks to perform sequence processing tasks has given rise to a wide range of models. An important consideration, however, is their ability to learn a particular data set. Here emphasis will be given to three recurrent network models: the Simple Recurrent Network (Elman 1990), the Real Time Recurrent Learning (RTRL) network (Williams and Zipser 1989) and the Gamma Model (de Vries and Principe 1992), as these will form the basis of comparative studies in chapters three and four of this thesis.

The Simple Recurrent Network has been applied across a range of sequence processing problems, particularly the processing of finite state grammars. Changes to

various aspects of the network reported by Kalman and Kwasny (1994) led to improvements in its ability to converge. These included changes to the architecture (adding skip connections) and to the learning algorithm (a new way to calculate the error). These gave rise to faster convergence. Interestingly, the Simple Recurrent Network has been shown to be capable of processing quite long sequences of widely varying lengths, even though the only information that is retained is the activation values of the hidden layer from the previous time step. This is in contrast to the Back Propagation Through Time model, where complete copies of the network are preserved for each element of the sequence.

Williams and Zipser (1989) report results of experiments using the RTRL network over a range of tasks. Although no statistics are given, they report that solutions to various problems are "readily found" by the network. Despite this, Zipser (1989) conducted research on ways to improve the performance of RTRL because of the high computational load that the network places on hardware (see page 36 for details of this modification). Again no statistical data was presented to back up claims that subgrouping an RTRL network can converge much faster than the original RTRL model. Comparative studies have shown that RTRL tends to perform rather poorly compared to other recurrent network models (see section 2.4 for further details).

Reports concerning the performance of the Gamma Model indicate that it is capable of learning quite complex tasks and benefits from receiving integrated data as input (Principe and Motter 1994). This enables faster learning and means that fewer hidden units need be used (although at a cost of expanding the size of the input layer). Principe, Kuo and Celebi (1994) argue that the recursive nature of the gamma memory structure gives rise to an additional parameter over recurrent networks which have tapped delay lines:

"[W]hen a [gamma network] is used...The angle between the desired signal and the memory space changes as a function of the feedback parameter. Therefore a memory

filter of a given order still has an extra parameter to decrease the difference between the desired signal and its orthogonal projection" Principe, Kuo and Celebi (1994) pp 336.

### **2.3 Attempts at classification.**

It is clear from section 2.1 that work on sequence processing using neural networks has produced a large collection of models. Furthermore the models are diverse in terms of their architectures, the learning rules that they use and the behaviours that they exhibit. Small wonder, then, that efforts have been made to create a set of rules by which different recurrent networks can be classified.

Mozer (1993) divides a neural network for sequence processing into two parts; the short term memory mechanism which captures those aspects of the input sequence that are needed to make accurate predictions and a feed forward structure that can make accurate predictions based on the input it receives from the short term memory structure. When designing a neural network for sequence processing Mozer believes that three distinct factors need to be taken into consideration: What is the architecture of the network (number of layers, units etc.)? What is the nature of the training algorithm? What form does the short term memory mechanism take?

Concentrating on the third factor in the above list, Mozer develops a taxonomy of short term memory structures based on the following three criteria:

*The form that the memory takes:* Mozer describes three different forms. The most simple form is where the memory mechanism is a buffer of size  $n$  which contains the  $n$  most recent inputs to the network, an example of this is the TRACE model of McClelland and Elman (1986) discussed above. Another form is the exponential trace memory, an example of which is Elman's SRN. Finally there is the gamma memory of de Vries and Principe. This gives us three different types of memory form.

*The content of the memory mechanism:* Although the memory mechanism must hold information about the sequence, the memory structure does not have to hold raw data. The sequence may be encoded into a new representation. Furthermore, this representation may also be transformed in some way. Further options for memory content can be seen by storing either the state of the network or its outputs. This gives us six different types of memory content.

*Memory adaptability:* The memory mechanism can be static, where the memory state is a predetermined function of some part of network activity. Alternatively the memory mechanism can be adaptive where, during learning, the network is able to select those aspects of the sequence which make the most contribution to correctly processing the sequence. This gives us another two parameters to classify a given memory structure.

In total, Mozer's classification system gives us thirty six different types of memory. Classifying various neural network models used in sequence processing tasks shows that the majority of work done so far has been concentrated on models which use a delay line. The gamma memory structure in particular has received little attention.

A different type of memory mechanism classification was advocated by Tsou and Back (1994). In total they advocate splitting neural networks for sequence processing into three categories:

*Models based on multi-layer perceptrons:* This involves using a feed forward network with Finite Impulse Response (FIR) filters. An example of this is the network described by Wan (1993).

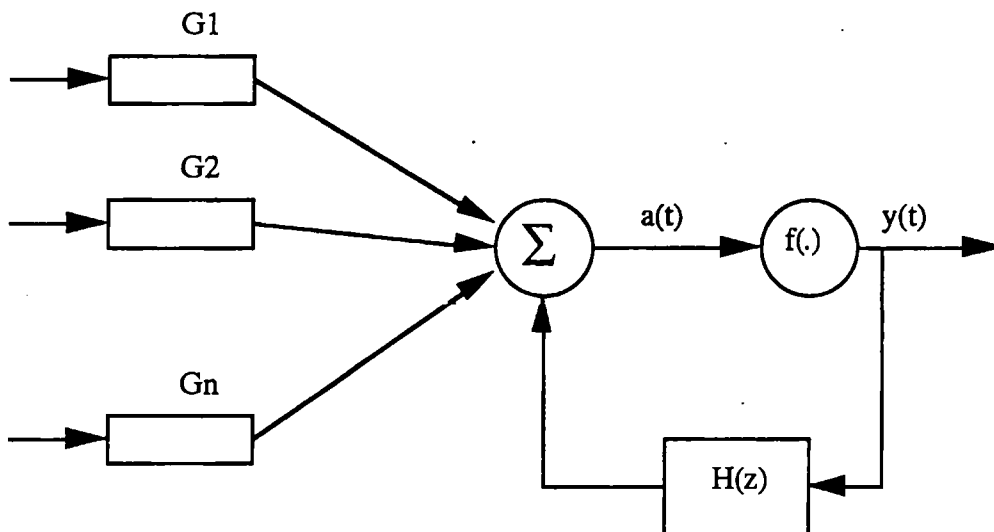
*Recurrent Networks:* As was stated earlier, these are networks which have feedback connections either within or between layers. Both the Simple Recurrent Network of Elman (1990) and the Real time recurrent learning network of Williams and Zipser (1989) fall into this category.

Finally, there is a group of models which combine some of the features of multi-layer perceptron based networks and recurrent networks. These are defined by

Tsoi and Back (1994) as *local-recurrent-global-feed forward (LRGF) models*. Such networks, as their name suggests will incorporate both feed forward and recurrent connections. Networks which fall into this category include the Back-Tsoi architecture, the gamma memory structure and memory neural networks. Tsoi and Back also establish their own criteria by which to judge LRGF models:

1. Does the model have the ability to be a universal approximator for a set of input-output mappings?
2. The model should be as simple (i.e. have as few units and connections) as possible.
3. What is the optimum form of feedback structure?
4. The model should be robust to structural perturbations.

Tsoi and Back also define a taxonomy for LRGF models, based on the type of synapse (simple or dynamic) and the feedback location (synapse, activation or output). In total this gives six different types of LRGF model. These can be summarised in a generalised LGRF architecture as shown in fig 2.10.



*Fig 2.10: Generalised LRGF architecture.  $G1, G2, \dots, Gn$  are local synapse feedback functions.  $H(z)$  is a local output feedback function.*

Perhaps the simplest classification system to date was suggested by Horne and Giles (1994). In their research, recurrent networks examined in their comparative study (see section 2.3) were divided into two categories: those with observable states and those without. A particular network was deemed to have observable states if its states can always be determined from observations of the input and output alone. Networks with hidden dynamics have states which are not easy to observe.

## **2.4 Other comparative studies.**

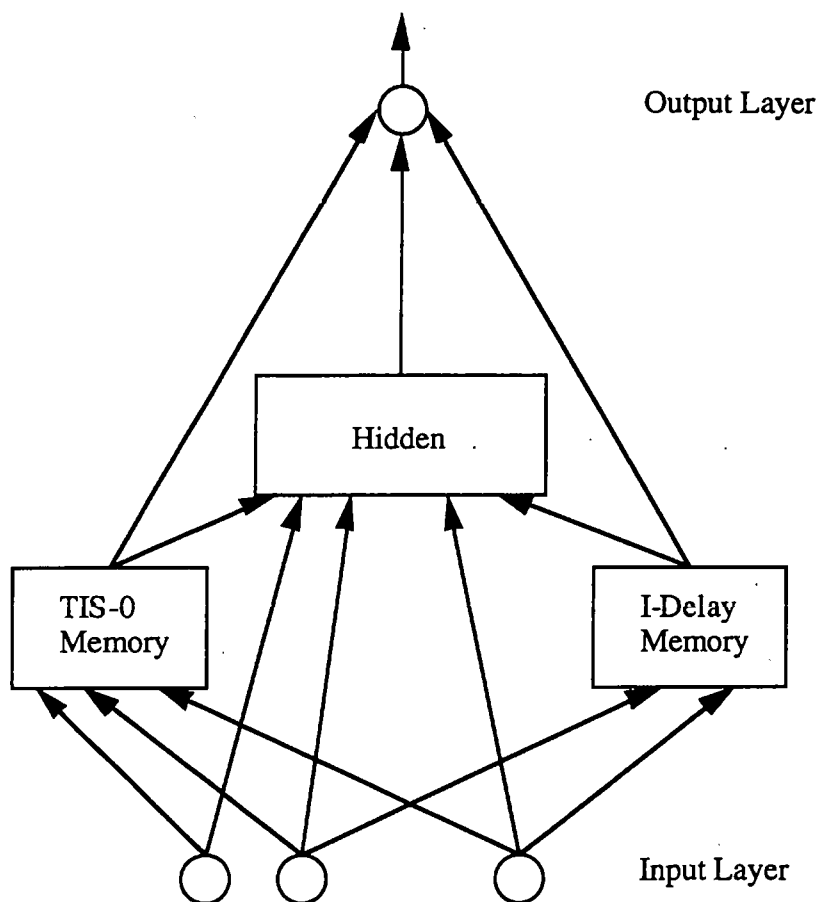
Associated with the methods of classification discussed in section 2.2, there are also comparative studies of the different recurrent network models. Additionally, some researchers have used comparative studies as a means to justify their particular model without attempting to fit this into any kind of overall framework.

Mozer (1993) carried out a detailed study of three recurrent networks, which differed according to the content of their short term memory mechanism:

1. An input memory mechanism (I), where the content was simply a copy of the inputs to the network.
2. A transformed input and state (TIS-0) mechanism, where a non-linear transformation is carried out over the current input and the current memory state.
3. A hybrid architecture which contained both memory mechanisms.

Mozer conducted the experiments by constructing a general architecture (see fig 2.11), eliminating memory mechanisms so that a particular network was tested.





*Figure 2.11 Generalised architecture developed by Mozer for use in a comparative study of recurrent networks over a time series analysis problem.*

The problem used to compare these networks was the dollar/Swiss franc exchange rate prediction problem used in the Santa Fe time series competition<sup>5</sup>. Each of the three networks were tested with different numbers of units per layer, different learning rates and learning rules. Interestingly Mozer found that the simplest model tested, the delay memory mechanism, performed as well as, if not better than, the more elaborate TIS-0 and hybrid models (see table 2.3).

---

<sup>5</sup> This data set is discussed further in chapter three pp 61.

Architecture	1 Minute Prediction (57773) data points
I-delay, 0 Hidden	.999
I-delay, 5 Hidden	.985
I-delay, 10 Hidden	.985
I-delay, 20 Hidden	.985
TIS-0	.986
Hybrid TIS-0 and I-delay	.986

*Table 2.3 Normalised Mean Squared Error for financial data series in comparative study carried out by Mozer (1993).*

Similarly, Tsoi and Back report the preliminary results of a comparative study in which they hope to demonstrate that LRGF models are the best neural networks for sequence processing tasks. In total four different networks were tested: The Back and Tsoi LRGF model, the Frasconi-Gori-Soda LRGF model, the RTRL model of Williams and Zipser and a feed forward network which incorporates time delayed inputs. These were tested on a speech recognition task. The results are shown in table 2.4.

Architecture	MSE	Variance
Back-Tsoi	0.0522	0.0097
Frasconi-Gori-Soda	0.0225	0.0041
Williams and Zipser	0.1803	0.1777
Feed forward Network	0.0299	0.0097

*Table 2.4 Mean squared error and Variance results from the Tsoi and Back (1994) comparative study of neural network performance over a speech prediction task.*

It is clear from the table that the LRGF model of Frasconi, Gori and Soda gives the best results, whilst the RTRL model of Williams and Zipser gives the poorest results.

As was stated earlier, some researchers have used comparative studies in order to demonstrate the usefulness of their own model. Principe and Turner (1994) compare the gamma model to a Finite Impulse Response (FIR) filter (see section 2.1) over a speech recognition task. Their results showed a superior performance by the gamma model (see table 2.5).

Memory	Epoch	Error
FIR	700	0.0321
Gamma	463	$2.81 \times 10^{-8}$

*Table 2.5 Comparative results between a gamma memory and a FIR filter over a speech recognition task. Table shows the minimum error and the number of epochs taken to reach this minimum. Taken from Principe and Turner (1994).*

A similar comparative study was performed by Principe and Motter (1994), who compared two different Time Delay Neural Network (TDNN) models with two networks which incorporated a gamma memory structure. The different attributes of the networks are summarised in fig 2.6.

	TDNN1	TDNN2	Gamma1	Gamma2
Input Layer	7	64	16	20+20
1st Hidden	14	14	14	6
2nd Hidden	6	6	6	0
Output	1	1	1	1

*Table 2.6 Description of Network topologies used in comparative study by Principe and Motter (1994).*

In Gamma2, the input layer consisted of two distinct layers, with an integrator between them, this accounts for the fewer number of hidden layers compared to the other networks in the study.

The networks were trained on their ability to identify the dynamics of a wind tunnel. Once again the gamma models outperformed the TDNN models (see table 2.7).

	TDNN1	TDNN2	Gamma1	Gamma2
Lowest MSE	0.02	0.004	0.005	0.003
Iterations	100,000	10,000	5,000	5,000

*Table 2.7 Results of the comparative study undertaken by Principe and Motter (1994). Mean Squared Error (MSE) and the number of iterations that the network was trained on are shown.*

Although TDNN2 and Gamma1 were able to learn the task, they did so whilst displaying significant oscillations. These were not exhibited by Gamma2.

Another comparative study which looked at the effect that different adaptive memory structures had on network performance was performed by Principe and Turner (1994). The details of their modifications are described in section 2.1 above. The modified Gamma model outperformed both the original gamma model as well as a time delay neural network model which has the memory layer restricted to the input layer. The task on which the networks were tested involved recognising spoken words. The results are shown in table 2.8.

Memory Type	Epoch	Error
TDNN	700	0.0321
Principe and Turner	463	$2.81 \times 10^{-8}$

*Table 2.8 Error scores for the Gamma memory structure proposed by Principe and Turner (1994) against a conventional Time Delay Neural Network. Epoch refers to the number of epoch needed to reach the error score shown in the third column.*

The comparative study of Horne and Giles (1994) is very wide ranging. Each network was tested over two different tasks: Learning a finite state machine and non-linear system identification. The number of weights and the number of states were kept approximately equal across all networks, since these factors may have a significant effect on learning<sup>6</sup>. The results of this study are shown in table 2.9a (for the finite state grammar) and table 2.9b (for the non-linear system identification problem).

---

<sup>6</sup> The number of weights that a network has is known to effect the ability of a feedforward network to generalise, but it is not known if this is also true of recurrent networks.

FSM	Architecture	Training Error	Testing Error	%P	W	S
RND	N&P	2.8 (4.4)	16.9 (8.6)	22	56	8
	TDNN	12.5 (2.1)	33.8 (4.1)	0	56	8
	Gamma	19.6 (2.4)	24.8 (3.2)	0	56	8
	First Order	12.9 (6.9)	26.5 (9.0)	0	48	6
	High Order	0.8 (1.5)	6.2 (6.1)	60	50	5
	Bilinear	1.3 (2.7)	5.7 (6.1)	46	55	5
	Quadratic	12.9 (13.4)	17.7 (14.1)	12	45	3
	Multi-layer	19.4 (13.6)	23.4 (13.5)	6	54	4
	Elman	3.5 (5.5)	12.7 (9.1)	27	55	6
	Local	2.8 (1.5)	26.7 (7.6)	4	60	20
FMM	N&P	0.0 (0.2)	0.1 (1.1)	99	56	8
	TDNN	6.9 (2.1)	15.8 (3.2)	0	56	8
	Gamma	7.7 (2.2)	15.7 (3.3)	0	56	8
	First Order	4.8 (3.0)	16.0 (6.5)	1	48	6
	High Order	5.3 (4.0)	26.0 (5.1)	1	50	5
	Bilinear	9.5 (10.4)	25.8 (7.0)	0	55	5
	Quadratic	32.5 (10.8)	40.5 (7.3)	0	45	3
	Multi-layer	36.7 (11.9)	43.5 (8.5)	0	54	4
	Elman	12.0 (12.5)	24.9 (7.9)	5	55	6
	Local	0.1 (0.3)	1.0 (3.0)	97	60	20

(a)

FSM	Architecture	Training Error	Testing Error	%P	W	S
	N&P	4.6 (8.4)	14.1 (11.3)	38	73	6
RND	TDNN	11.7 (2.0)	34.3 (3.9)	0	73	6
	Gamma	19.0 (2.4)	25.2 (3.1)	0	74	6
	First Order	12.9 (6.9)	26.5 (9.0)	0	48	6
	High Order	0.3 (0.5)	4.6 (5.1)	79	74	6
	Bilinear	0.6 (0.9)	4.4 (4.6)	55	78	6
	Quadratic	0.2 (0.5)	3.2 (2.6)	83	216	6
	Multi-layer	15.4 (14.1)	19.9 (14.4)	16	76	6
	Elman	3.5 (5.5)	12.7 (9.1)	27	55	6
	Local	13.9 (4.5)	20.2 (5.7)	0	26	6
FMM	N&P	0.1 (0.8)	0.3 (1.4)	97	73	6
	TDNN	6.8 (1.7)	16.2 (2.9)	0	73	6
	Gamma	9.0 (2.9)	14.9 (2.8)	0	73	6
	First Order	4.8 (3.0)	16.0 (6.5)	1	48	6
	High Order	1.2 (1.7)	25.1 (5.1)	31	74	6
	Bilinear	2.6 (4.2)	20.3 (7.2)	21	78	6
	Quadratic	12.6 (17.3)	26.1 (12.8)	13	216	6
	Multi-layer	38.1 (12.6)	42.8 (9.2)	0	76	6
	Elman	12.8 (14.8)	27.6 (10.7)	8	55	6
	Local	15.3 (3.8)	22.2 (4.9)	0	26	6

(b)

*Table 2.9 Results of the comparative study performed by Horne and Giles (1994) using the data generated from a finite state grammar. Table (a) shows results when the networks have an approximately identical number of weights. Table (b) shows results when the networks have an approximately identical number of state variables. %P denotes the number of trials for which the training set was learned perfectly. W denotes the number of weights. S denotes the number of states. Note that the gamma kernels used in this study were not adaptive.*

Architecture	Fixed weights	Fixed states
N&P	0.101	0.067
TDNN	0.160	0.165
Gamma	0.157	0.151
First Order	0.105	0.105
High Order	1.034	1.050
Bilinear	0.118	0.111
Quadratic	0.108	0.096
Multi-layer	0.096	0.084
Elman	0.115	0.115
Local	0.117	0.123

*Figure 2.10: Mean squared error on a test signal for the non-linear system identification problem. The column denoted "Fixed weights" is the results when all networks had a similar number of weights. The column denoted "Fixed states" is the results when all networks had a similar number of states.*

## 2.4 Conclusion

The search for neural network models capable of solving sequence processing problems has spawned a large number of diverse architectures and algorithms. The predominant class of neural network models used by researchers are known as recurrent networks. However, choosing this one class only slightly clarifies the picture, since the choice is still large and diverse. This has produced two closely connected avenues of research: classification and comparison.

These two research efforts have yielded some useful information. Classification studies have shown that recurrent networks can be defined according to a particular taxonomy. The more variables in the taxonomy leads to a more fine grained classification. However, in comparison to the amount of research done on



developing new networks and the modification of existing networks, the number of systematic comparative studies performed is relatively small.

Of the comparative studies that have been performed, some have been attempts to prove the effectiveness of the researcher's own model against other existing architectures and/or algorithms. There has been some work, however, that can be defined as "purely comparative". Such studies have attempted to identify the most effective recurrent networks so that future research (either pure or applied) can be concentrated on them. The aim of this thesis is to pinpoint the most effective form of recurrent network in order to facilitate this research.

## **Chapter Three. A Comparative Study of Three Recurrent Network Models**

### **3.1 Overview and Rationale**

In chapter two we examined the numerous attempts to design neural networks capable of performing sequence processing tasks, together with attempts to classify and compare them. Although many of the comparative studies done so far have provided useful information, they have often been limited in that either one data set was used and/or only a particular type of neural network has been tested. Since the term 'sequence processing' covers a wide range of phenomena, it will be useful to compare the performance of different recurrent networks over more than one task. It is worth using multiple data sets for the following reasons:

*Attributes of the sequence:* Different sequences have different properties. An element of a sequence at a single time step may only be influenced by events in the recent past. Alternatively longer term factors may well play a part. If for a given sequence both short and long term factors are significant, how important are they and what is the nature of the interaction (if any) between them? Are the sequences of the same or similar length or do they vary quite widely? Hence the design of a network for a particular task will be affected by what is known about the attributes of the sequence. If this information is known and easily defined then the type of short term memory mechanism can be much more easily specified. Alternatively if the information is less known or less easily defined then we would seek to use the most powerful short term memory mechanism available, since a powerful mechanism may be what is needed.

*Attributes of the network:* Are different recurrent networks suited to particular types of sequence? Or is there one type which significantly outperforms the others over a wide range of sequence types? The ability of the network to perform a given sequence processing task is determined by the following factors:

- The nature of the short term memory mechanism: Is the mechanism powerful enough to retain information about the sequence of data of sufficient quantity and quality to ensure that learning can take place?
- The level of connectivity of the network: Does the network have a state space which is rich enough to form an internal representation of the sequence?
- The learning rule: Does the learning rule allow the network to traverse the state space in such a way that it can avoid local minima but still find the global minimum.

This chapter describes the details of a comparative study of three recurrent network models over four sequence processing tasks. Section 3.2 gives the details of the different tasks. The networks which were tested can be found in section 3.3. Finally, section 3.4 describes the results of both modifying the internal parameters of each network and the effect that this has on network performance, as well as the comparative study itself.

## 3.2 The Tasks

A number of different types of time series were used to measure network performance, so that comparing the networks over a range of different tasks will give a better idea of their overall capabilities.

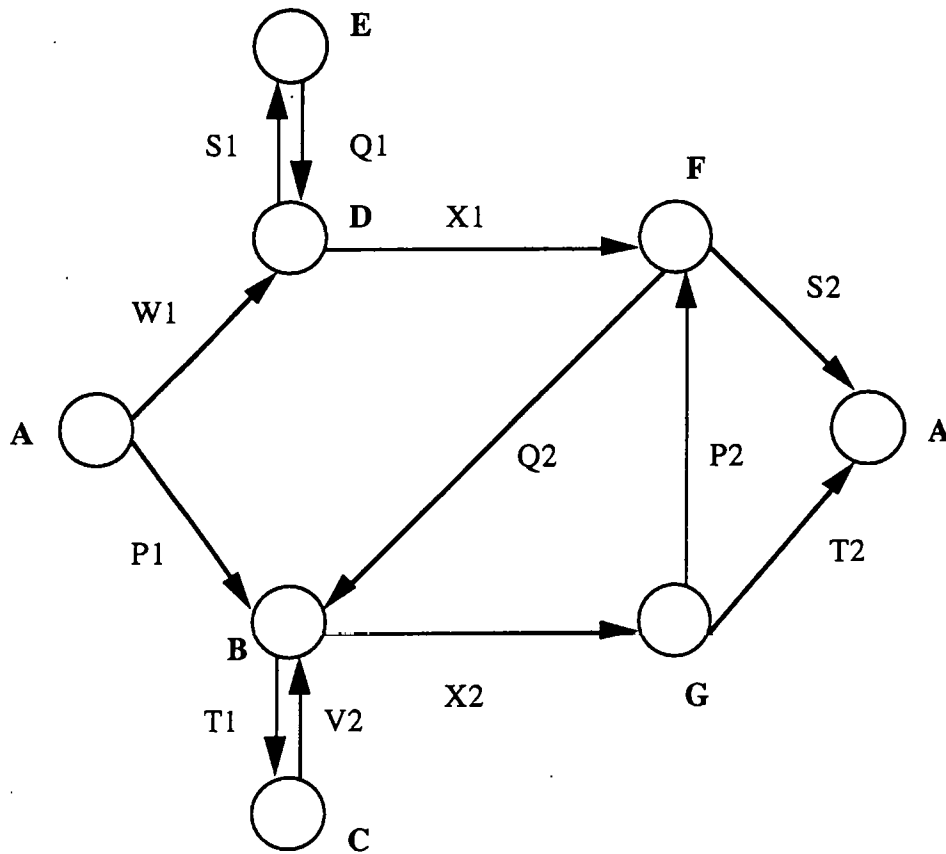
1. A near replication of the letter in word prediction task described in Elman (1990). The difference is that Elman used a thirteen word lexicon to generate his data sets whereas a fifteen word data set was used in this study. The learning data was a non-grammatical sequence of one thousand words each of which was chosen at random from a fifteen word lexicon. The test data was a sequence of twenty words drawn

Time	Input	Desired Output	Output = solution at time
t=1	0 0	n/a	n/a
t=2	1 0	n/a	n/a
t=3	0 1	0	t=1
t=4	1 1	1	t=2
t=5	0 0	1	t=3

*Table 3.2: Sample data for the XOR with two step delay task, showing the memory needed by the network to solve the task.*

Unlike the Elman task, the network should have a consistently low error score as soon as enough information is available (in table 3.2 for example the network should have a low error score from time t=3 onwards).

3. A sequence generated from a simple finite state grammar, as described in Cleeremans and McClelland (1989 ). The grammar consisted of eight nodes and a total of twelve arcs connecting them (see figure 3.1). The learning data consisted of one hundred and fifty complete traversals (of varying length) of the finite state grammar. The test data consisted of ten complete traversals, again of varying length.



*Fig 3.1. A finite state grammar used to generate data set three (see above), which is identical to the one used in Cleeremans and McClelland (1993). Nodes are labelled with bold letters, connections with plain text.*

A finite state machine in this case consists of nodes A-G and a series of connections P,Q,S,T,W,X. Each of these letters is applied to two nodes. The data set is generated by traversing the network and noting down the letter associated with the particular node. Each of these letters was assigned a number one through to six respectively. The input to the network took the binary form of each number. This is shown in table 3.3.

Letter	Number	Binary Form
P	1	001
Q	2	010
S	3	011
T	4	100
W	5	101
X	6	110

*Table 3.3: Representation of nodes for the Finite State Grammar task.*

If there is a choice of steps from a particular node, then the next node in the sequence is chosen randomly. The network is presented with the traversal one node at a time, and has the next connection label in the sequence as the desired output. For the finite state machine in figure 3.1 for example, the traversal A-B-C-B-G-F-A would produce the training data shown in table 3.4:

Input	Desired Output	Input to Network	Desired Output of Network
P	T	001	100
T	W	100	101
W	X	101	110
X	P	110	001
P	S	001	011

*Table 3.4 Example of training data for letter in Finite State Grammar task.*

This data set has attributes similar to those of the Elman letter in word prediction task described above. However there is a significant difference in that the

potential length of a sequence is much greater, requiring the network to have a more powerful and flexible short term memory mechanism.

4. One of the classic problem areas in time series learning and prediction was examined: financial data series, formed by the exchange rate between Swiss Franc and US dollars.\* This data set was used in the SantaFe institute's time series competition and the problem is a classic one in time series literature. It is also a test which has been used in comparative studies of recurrent networks and other time series prediction methods. The learning and test data were a single portion of a ten thousand point series, the last one hundred points formed the test data, the rest formed the learning data. This data set has the following attributes:

- The exchange rate is the product of both short term and long term factors, which can be subjective (the mood of the currency dealers) as well as objective (the health of the US and Swiss economies and the nature of trade between them).
- The network will need a powerful short term memory mechanism and a rich architecture capable of reflecting the complex nature of the sequence in its state space.

The SantaFe data was modified into a form readable by the network by converting the data into a binary representation.

\* see Predicting the future and understanding the past, A. Weigend and N. Gershenfeld (Eds), Addison-Wesley, 1993.

### 3.3 The Networks<sup>1</sup>

In this comparative study, three different recurrent network models were examined: The simple recurrent network (SRN) used by Elman (1990), the Real Time Recurrent Learning (RTRL) model devised by Williams and Zipser (1989) and the Gamma memory model of deVries and Principe (1992). Each of the three networks were constructed as described in these papers, with none of the later modifications that have been suggested by some researchers (see chapter four). Table 3.5 shows the different sizes of each network over the different tasks described in section 3.2 above.

---

<sup>1</sup>The networks examined in this chapter are described in sections 2.1.4 (Simple Recurrent Network), 2.1.7 (Real Time Recurrent Learning) and 2.1.8 (Gamma).



	Simple Recurrent Network	Real Time Recurrent Learning	Gamma Memory	Input Layer Size	Output Layer Size
2-delay XOR	Hid = 4	RTRL = 4	Hid = 4	2	1
Letter in Word P.T	Hid = 20	RTRL = 20	Hid = 20	5	5
Finite State Grammar	Hid = 20	RTRL = 20	Hid = 12	3	3
Dol - SF Ex Rate	Hid = 20	RTRL = 20	Hid = 20	10	10

*Table 3.5: Architecture of the models tested, giving number of units in the "hidden" layer for each network. The gamma memory layer  $k$  is below the hidden layer (see fig 3.2c). The architecture is the same for gamma kernel size  $k=1$  and  $k=2$ . The output "layer" for the RTRL network is a subset of the RTRL layer.*

These networks are also shown in figure 3.2

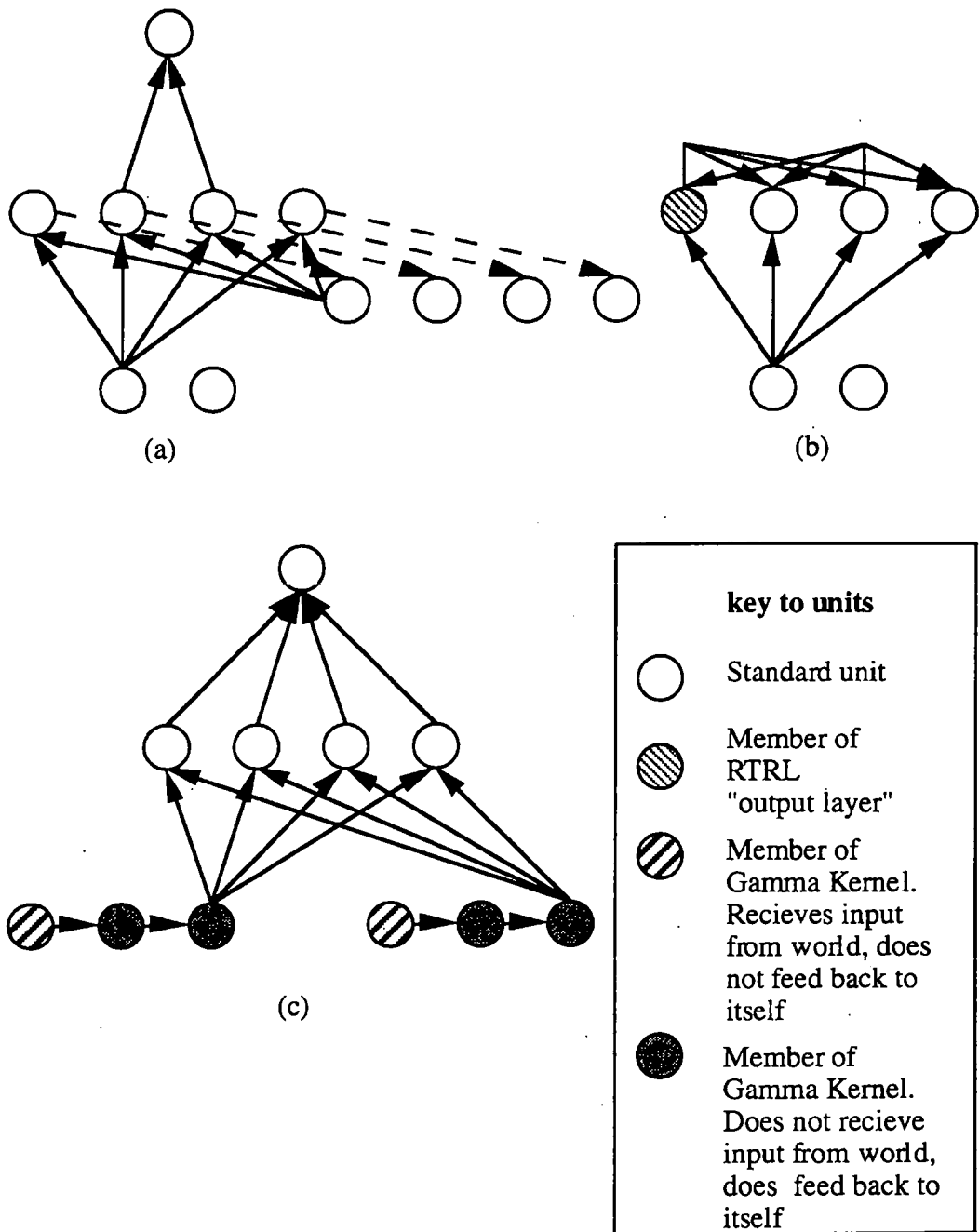


Figure 3.2. Illustrations of architectures used for the XOR with two step delay problem: (a) A Simple Recurrent Network, (b) an RTRL Network and (c) a Gamma memory model with kernel size = 2. Some connections have been deleted to aid clarity. Connections marked by solid lines are trainable. Connections marked by dashed lines are not trainable and have a fixed value = 1.

### 3.4 Experimental Variables

Each network model (SRN, RTRL, Gamma) was tested using the following as experimental variables: Different numbers of hidden (or RTRL or Gamma) units and different learning rates (1,2 or 4). Each combination was tested ten times to measure sensitivity to initial conditions. For the purposes of the comparative study, results from networks with the combination of units which gave the best performance were selected. Two distinct gamma networks were run; one with a kernel of size one, the other with a kernel of size two.

All the experiments were run using the Neuralworks simulation package, supplied by Scientific Computing, on a Sun workstation. The learning algorithms for the RTRL and Gamma networks, as well as the summation function for the Gamma network were developed in C using the User Defined Neuro-Dynamics package, which was supplied by the same company. The code for these algorithms is shown and described in appendices one and two. The RTRL algorithm is based on the equations used in Williams and Zipser (1989). The Gamma model is based on the equations used in deVries and Principe (1992).

## 3.5 Results

### 3.5.1. The effects of modifying internal parameters on Learning

Although the three networks described in this study are diverse in terms of their architectures and learning rules, it is still possible to describe general behaviours common to all of them.

Increasing the size of the hidden layer led to an increase in the ability of the network to learn the pattern set, with a subsequent increase in ability to predict the test data. Furthermore the fluctuations observed under certain conditions were more pronounced as the size of the hidden layer was increased. Once the hidden layer size had been increased to a size such that learning could take place, increasing it still further had no effect on network performance, other than increasing the time the network took to cycle through the pattern set.

An example of this can be seen when we examine the performance of the gamma model over the letter in word prediction task. In this experiment we are looking to replicate Elman's finding that the error is high at the start of the word and decreases as more of the word becomes known (with a subsequent decrease in ambiguity). Typical test results are shown in figure 3.3. In this example the two networks were identical in every <sup>other</sup> respect, with a gamma kernel size of two and a learning rate of one.

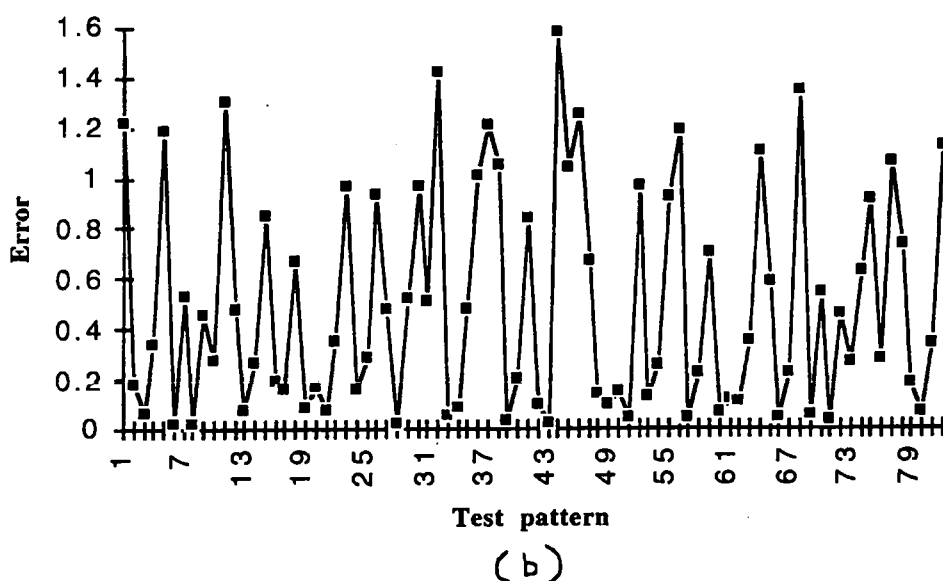
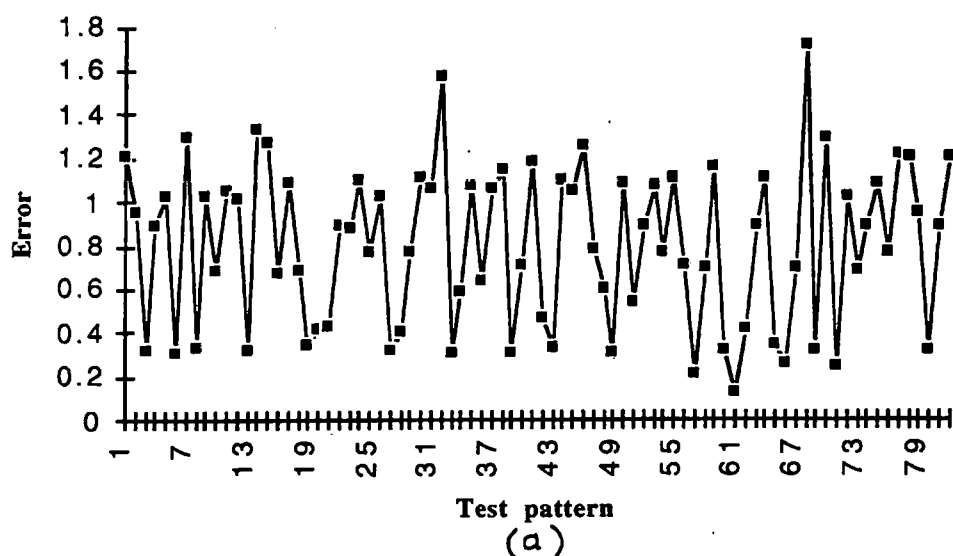


Figure 3.3 (a) Test data for the Gamma model with five hidden Units over the Letter in Word Prediction task. b) Test data for the Gamma model with twenty hidden Units over the Letter in Word Prediction task

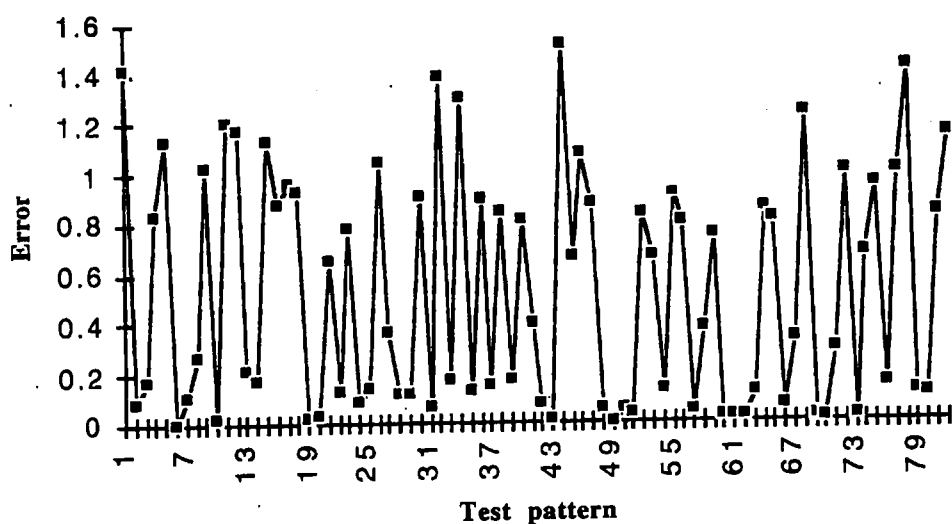
Note that when the gamma network with five hidden units is tested, the overall error is not as low, and the sudden dips in error over the course of a word are not as sharply marked.

The behaviours which were observed with the changing of hidden layer size were also encountered with the Gamma layer. Often the network could learn with a

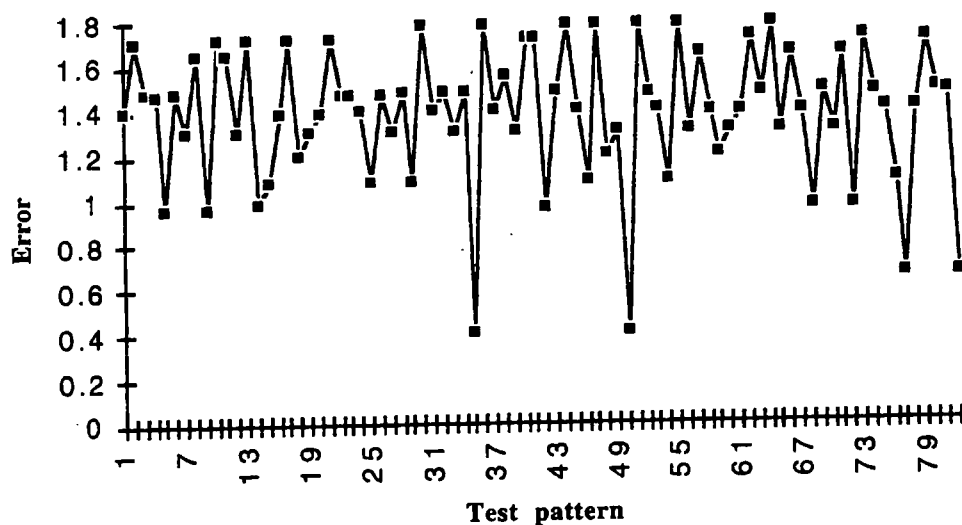
kernel size  $K=1$ . Increasing  $K$  once the network could perform a particular task had no effect on network performance.

With regard to learning coefficients, the network is most likely to successfully learn the data when the learning rate is small. As the size of the coefficient increases, the speed at which the network learns may well increase, but at the same time learning becomes more unstable and of poorer quality. This was most marked with the Gamma model. So that, when for example  $C=2$ , the network learns on some trials, but not on others. There was also a tendency for the error value to increase and decrease, rather than simply decreasing over time. If  $C$  is too large, the network will not learn at all. Similarly with the RTRL network on the XOR problem with two step delay, the increase in learning speed that came with increasing the size of the learning coefficient brought increased instability.

Figure 3.4 shows the performance of two different simulations with the same parameters. Both simulations were run with a kernel size two with twenty hidden units. Figure 3.4(a) shows the results on the test data from a network using a learning rate of one. Over the ten trials the sort of behaviour predicted by Elman (1990) can be clearly seen. Conversely in figure 3.4(b) the results on the test data from a network using a learning rate of two shows a set of behaviour less faithful to Elman's results. The reason for this is that when the learning rate was two, the network learned the task on some simulations but not on others.



(a)



(b)

Figure 3.4: Two sets of results from the gamma model performing Elman's letter in word prediction task.

Table 3.5 gives details of the architectures used in terms of the number of units. Whilst other combinations were tried, the architectures that gave the best results (or the smallest architecture if results did not vary over network size) are the

ones used in tables 3.5 and 3.6. In table 3.5, Inp is the number of input units; Hid is the number of hidden units and Out is the number of output units.

Note that the "output" units for the RTRL network are in fact a subset of the hidden units. For the gamma memory network two models were used, one with a kernel size of 2 ( $K=2$ ), the other with a kernel size of 1 ( $k=1$ ). To calculate the total number of units in the gamma layer, use the formula:  $U = \text{Inp} * (k+1)$ , where U is the number of gamma units. The hidden layer value for the SRN is the size of one hidden layer, the second hidden layer (i.e. the context layer) is of the same size.

### 3.5.2. Comparative Study Results

The RMS error scores for each of the networks averaged over ten runs are presented in table 3.6.

	Simple Recurrent Network	Real Time Recurrent Learning	Gamma Memory $K=1$	Gamma Memory $K=2$
2-delay XOR	0.542	0.240	0.498	0.205
Letter in Word Prediction	0.648	1.011	0.702	0.713
Finite State Grammar	0.669	0.816	0.683	0.675
Dollar - Swiss Franc Exchange Rate	1.080	1.129	1.080	1.100

*Table 3.6: RMS error scores for each of the networks across the test data of the four problems described above.*



### 3.5.3 Statistical Analysis of Results

In order to get a better appreciation of the performance of the networks over the various sequence processing tasks, statistical analysis of the results was carried out. The results of this are shown in the following tables:

Table 3.7 For the XOR with two step delay.

Table 3.8 For Elman's letter in word prediction task.

Table 3.9 For the finite state grammar.

Table 3.10 For the financial data series.

	SRN	RTRL	Gamma K=1	Gamma K=2
Mean	0.542	0.240	0.498	0.205
Standard Deviation	0.055	0.055	0.012	0.152
Range: Min	0.524	0.219	0.485	0.007
Max	0.706	0.403	0.523	0.426
Confidence Limits	0.542±0.039	0.240±0.039	0.498±0.003	0.205±0.034

*Table 3.7 Summary statistics for the four recurrent networks tested over the XOR with two step delay task. Figures shown represent the mean, standard deviation, range and confidence limits for the root mean squared error over the test data.*

As well as the statistics shown above, an analysis of variance (ANOVA) together with the Newman-Keuls test to check for significant differences between

pairwise comparisons. These revealed that the networks differed significantly on their performance on this task. The four networks could be split into two groups: The gamma network with a kernel size of two and the RTRL network in one group significantly doing better than the gamma network with a kernel size of one and the Simple Recurrent Network in the other group. There was no significant difference within these two groups.

	SRN	RTRL	Gamma K=1	Gamma K=2
Mean	0.648	1.011	0.702	0.713
Standard Deviation	0.018	0.095	0.011	0.134
Range: Min	0.630	0.857	0.683	0.639
Max	0.683	1.215	0.717	1.095
Confidence Limits	0.648±0.004	1.011±0.021	0.702±0.003	0.713±0.031

*Table 3.8 Summary statistics for the four recurrent networks tested over the Elman letter in word prediction task.*

Analysis of variance again revealed significant differences between the four networks. The RTRL network performed significantly worse than the other three networks tested ( $p < 0.01$ ). Analysis using the Newman-Keuels test showed that there was no significant difference between the other three networks over this task.

	SRN	RTRL	Gamma K=1	Gamma K=2
Mean	0.669	0.816	0.683	0.675
Standard Deviation	0.001	0.001	0.001	0.019
Range: Min	0.655	0.802	0.681	0.651
Max	0.688	0.828	0.685	0.828
Confidence Limits	0.669±0.001	0.816±0.001	0.683±e e<0.001	0.675±0.004

*Table 3.9 Summary statistics for the four recurrent networks tested over the Finite state grammar task.*

Analysis of variance showed that the RTRL network performed significantly worse than the other three networks ( $p<0.01$ ). There was also a less significant difference ( $p<0.05$ ) between the simple recurrent network and the gamma network with a kernel size of one.

	SRN	RTRL	Gamma K=1	Gamma K=2
Mean	1.080	1.129	1.080	1.100
Standard Deviation	0.013	0.045	0.008	0.008
Range: Min	1.062	1.095	1.066	1.086
Max	1.106	1.247	1.090	1.116
Confidence Limits	1.080±0.003	1.129±0.011	1.080±0.002	1.100±0.002

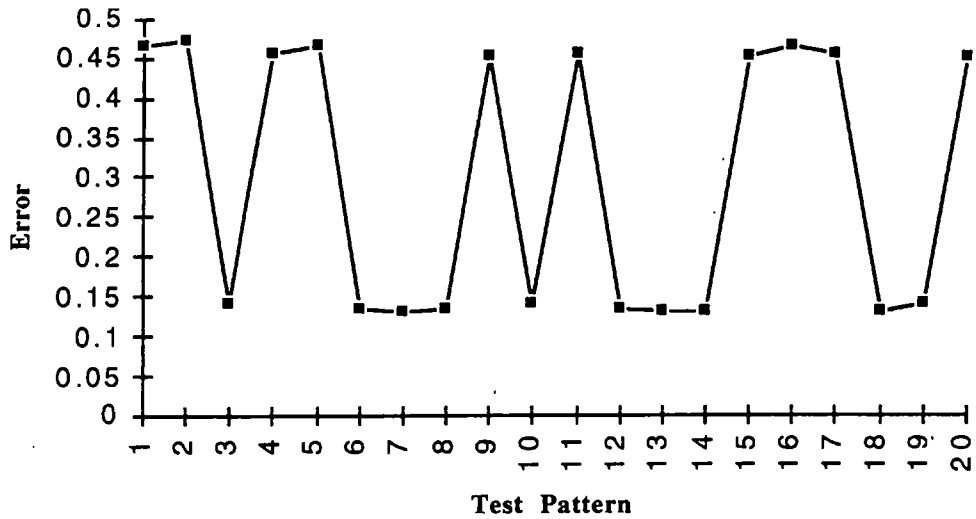
*Table 3.10. Summary statistics for the four recurrent networks over the Dollar to Swiss Franc exchange rate data*

Analysis of variance for this data revealed that the RTRL network performed significantly worse than the other three networks. Although the difference was less pronounced for the gamma model with a kernel size of two ( $p < 0.05$ ) than for the other two models ( $p < 0.01$ ). There were no significant differences between the other three models.

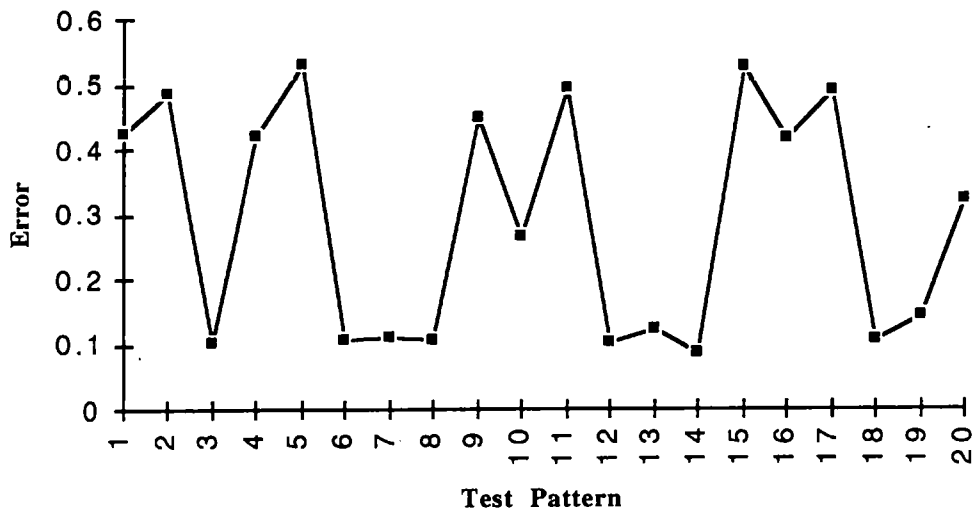
### 3.5.4 Continuous XOR with Two Step Delay

All three networks proved to be capable of performing this task, which proved to give the best results across all tasks. The Gamma model with kernel  $K=2$  proved to be the most successful at learning this task, followed by the RTRL and the Simple Recurrent Network. Although the error score for the Simple Recurrent Network appears to be fairly low, it is worth pointing out that the error score over the test set shows that the error is quite low on some of the data, but not on others. Increasing the size of the hidden layer to ten units also failed to bring about an

improvement in performance. The mean error for a Simple Recurrent Network with ten hidden units is 0.291. Performance on the test sequence is shown in figure 3.5:



(a)



(b)

Figure 3.5: (see previous page) Graph depicting the average performance of ten presentations of test data to ten different learning trials of the Simple Recurrent Network on the XOR with two step delay task. Graph (a) shows the performance of the network using four hidden units. Graph (b) shows the performance of the network using ten hidden units.

Interestingly, figure 3.5 shows that the addition of hidden units does not really change the performance of the network, even though the state space of the network is made much more complex. It is worthwhile noting that the Simple Recurrent Network only receives both the present input and a copy of hidden layer activations from the previous time step. It may well be the case that it is the nature of the memory mechanism that is the problem, since the RTRL network with four units proved to be more than capable of solving this problem.

### **3.5.5 The Letter in Word Prediction Task**

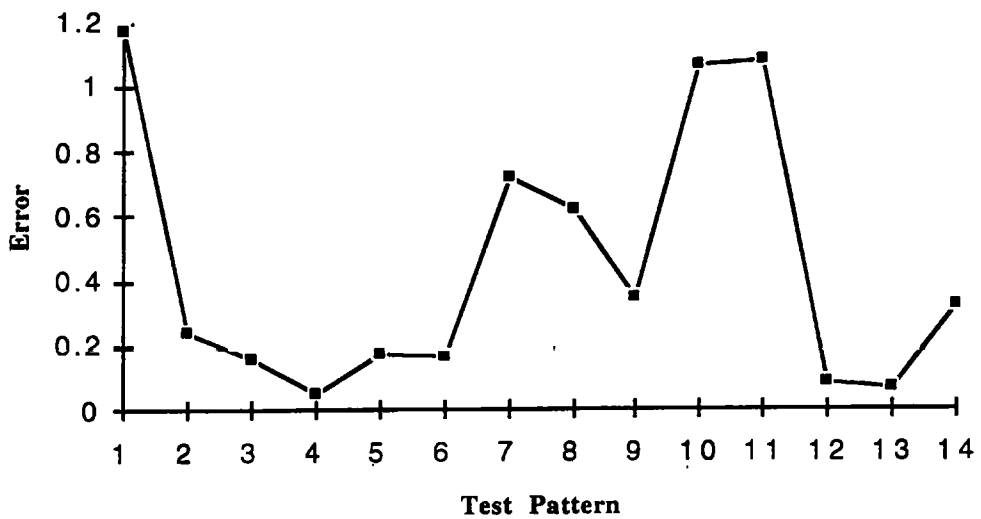
Both the Elman and the Gamma network were able to learn the data. On the test data the Gamma network exhibited the same behaviour as the Elman network; i.e. error is highest at the start of the word and decreases as more of the word is held in the network's short-term memory (see the description of the problem made by Elman for further details). Both performed significantly better than the RTRL network, which seemed unable to perform this particular task. Furthermore this inability was not affected by the addition of RTRL units, which only served to slow down even further the training process. Table 3.8 provides a numerical summary of these findings. One interesting point is that whilst the gamma model was being tested, one of the trials failed to find a satisfactory solution with a subsequent failure to decrease error significantly during learning. This sets the Gamma Model apart from the Simple Recurrent Network which learned the data successfully across all ten trials. This may account for the significant difference between the two ( $p < 0.05$ ).

Note, however, that although the mean error scores are higher than for the XOR with two step delay task, what we are looking for here is the ability of the network to replicate the behaviour of Elman's original study: that error should be high at the start of a word and decrease as ambiguity about the word decreases. To demonstrate this it is useful to look at a portion of the test data consisting of three words and to examine the behaviours of each network in turn. The portion consists of the words [ANIMAL, DOG, BOUND] and breaks down into the following sets of input / desired output pairings:

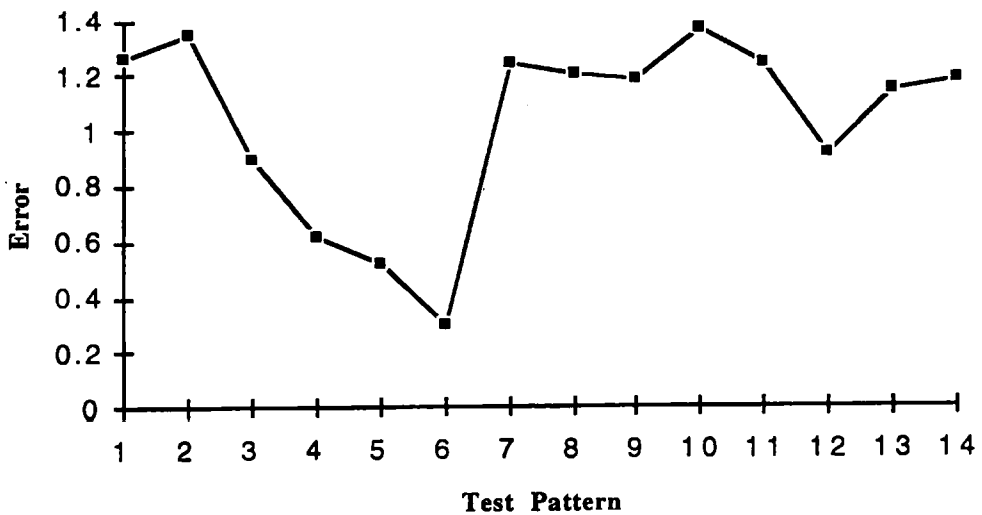
Pattern Number	Input	Desired Output
1**	E	A
2	A	N
3	N	I
4	I	M
5	M	A
6	A	L
7**	L	D
8	D	O
9	O	G
10**	G	B
11	B	O
12	O	U
13	U	N
14	N	D

*Table 3.11 A portion of the test data from the letter in word prediction task.*

Note that in table 3.11 those patterns marked \*\* are the transition points between words and are, according to Elman's model, where a sudden increase in the error score is to be expected.



(a)



(b)

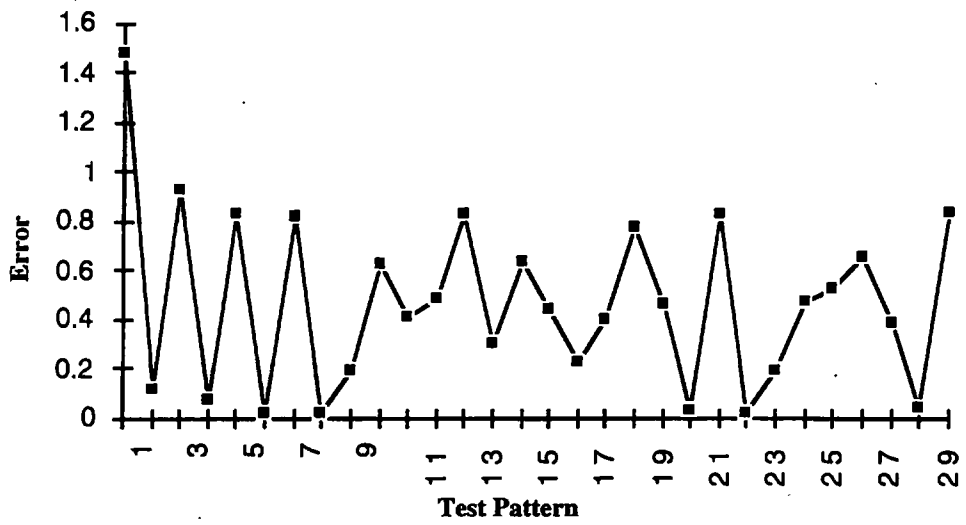
Figure 3.6 (a) Performance of the Gamma model over the portion of the pattern set described in table 3.11 (b) Performance of the RTRL network over the same data.



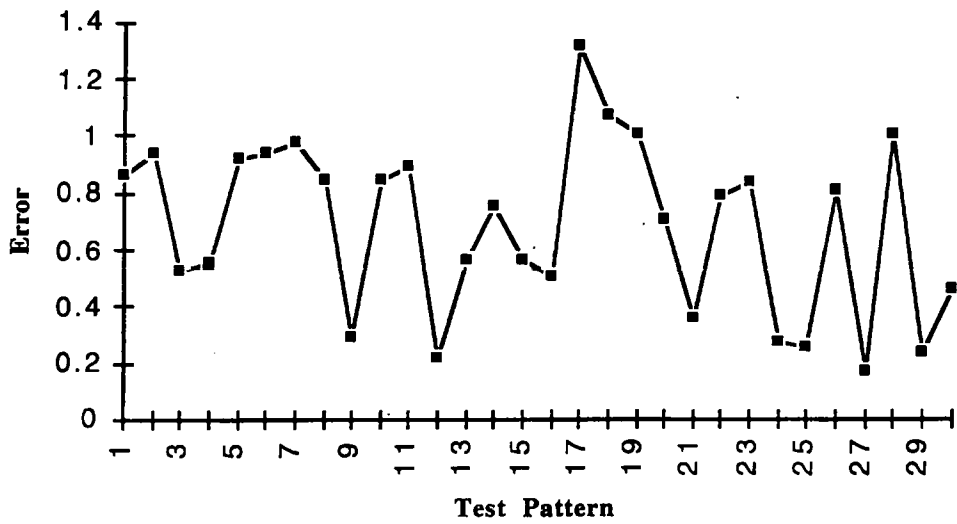
According to Elman's original findings high error rates are to be expected for patterns 1,7 and 10. This is clearly shown to be the case for the Gamma model. The same behaviour can also be vaguely seen in the RTRL model, but is much less clear. Note that even when the ambiguity is much reduced, the RTRL model still seems to have a great deal of difficulty in predicting the next letters in the sequence.

### **3.5.6 Learning a Finite State Grammar**

The results with this task were similar to those shown in the Letter in Word Prediction Task. Interestingly the RTRL network performed better on this task than on the Letter in Word Prediction Task. However its performance was still worse than both the SRN and the gamma networks. Both the Gamma Model and the Simple Recurrent Network learned the data successfully across all ten trials. The difference in performance between the Simple Recurrent Network and the RTRL network is shown in figure 3.7 (see overleaf).



(a)



(b)

Figure 3.7: Performance of (a) A simple recurrent network and (b) an RTRL network over the first thirty points of the Finite State Grammar test data.

### **3.5.7 Dollar to Swiss Franc Exchange Rate**

This problem proved to be the most difficult for all the networks used in this study. The behaviours shown by the networks were similar in that they were all fairly ineffective at learning the training data or predicting the test data. However, statistical analysis of the results showed significant differences in performance between the networks.

### **3.6 Some Observations on the $\mu$ Parameter**

As described above, the gamma memory has a parameter  $\mu$  which is used to vary the parameters of the memory, so that depth and resolution of memory can adapt to a level appropriate for a particular task. The different values of  $\mu$  evolved over the tasks used in this study are shown in table 3.12 (see overleaf)

	Gamma Memory K=1	Gamma Memory K=2
2-delay XOR	Max = 0.755 Min = 0.569 Mean = 0.667 M. Depth = 1.499 RMS Error = 0.248	Max = 1.049 Min = 1.037 Mean = 1.042 M. Depth = 1.919 RMS Error = 0.023
Letter in Word Prediction	Max = 1.278 Min = 0.711 Mean = 0.998 M. Depth = 1.002 RMS Error = 0.493	Max = 1.398 Min = 0.63 Mean = 0.962 M. Depth = 2.078 RMS Error = 0.525
Finite State Grammar	Max = 1.000 Min = 1.000 Mean = 1.000 M. Depth = 1 RMS Error = 0.467	Max = 1.302 Min = 0.551 Mean = 0.921 M. Depth = 2.171 RMS Error = 0.456
Dollar - Swiss Franc Exchange Rate	Max = 1.249 Min = 0 Mean = 0.512 M. Depth = 1.953 RMS Error = 1.166	Max = 1.207 Min = 0 Mean = 0.443 M. Depth = 4.509 RMS Error = 1.209

*Table 3.12: Maximum (Max), minimum (Min), mean values (Mean) of gamma memory parameter  $\mu$ , mean memory depth (M. Depth) and RMS Error after presentation of learning data.*

### 3.7. Discussion

#### 3.7.1 Behaviour of Recurrent Networks in General

Although this is a comparative study, where the main focus of attention is to see what differences exist between different types of recurrent network, an opportunity also arises to see what different types of recurrent network have in common. This will also be of help in assessing which features of recurrent network architectures or learning rules give rise to improved performance.

Obviously the size of the hidden layer is of critical importance. In order to learn a given data set, the hidden layer of a recurrent network must be sufficiently complex to form an internal representation of the data. This complexity is in terms of both the number of units in the hidden layer and the level of connectivity. In the terminology of dynamical systems a neural network can be viewed as a state space of  $N$  dimensions, where  $N$  is the number of connections within the network. A point  $P$  within that state space represents a matrix of connection strengths, which will embody the desired behaviour of the network to a greater or lesser degree.

If during learning the connection strengths of a network are such that some of the features of the data set have been modelled (i.e. the error is lower than at the start of learning but still falls short of the desired output) then the network lies in a local minimum. Conversely, if the connection strengths of a network are such that all of the features of the data set have been modelled (i.e. the error is zero or within some permitted range) then the network lies in a "satisfactory" minimum<sup>2</sup>. The number of "satisfactory" minima will depend on the number of connection strength sets that

---

<sup>2</sup> There are two types of "satisfactory" minima: Firstly the global minima of the state space, which represents a set of connection strengths which represent the lowest possible error value. Secondly those local minima which represent a set of connection strengths which represent a tolerable error value.

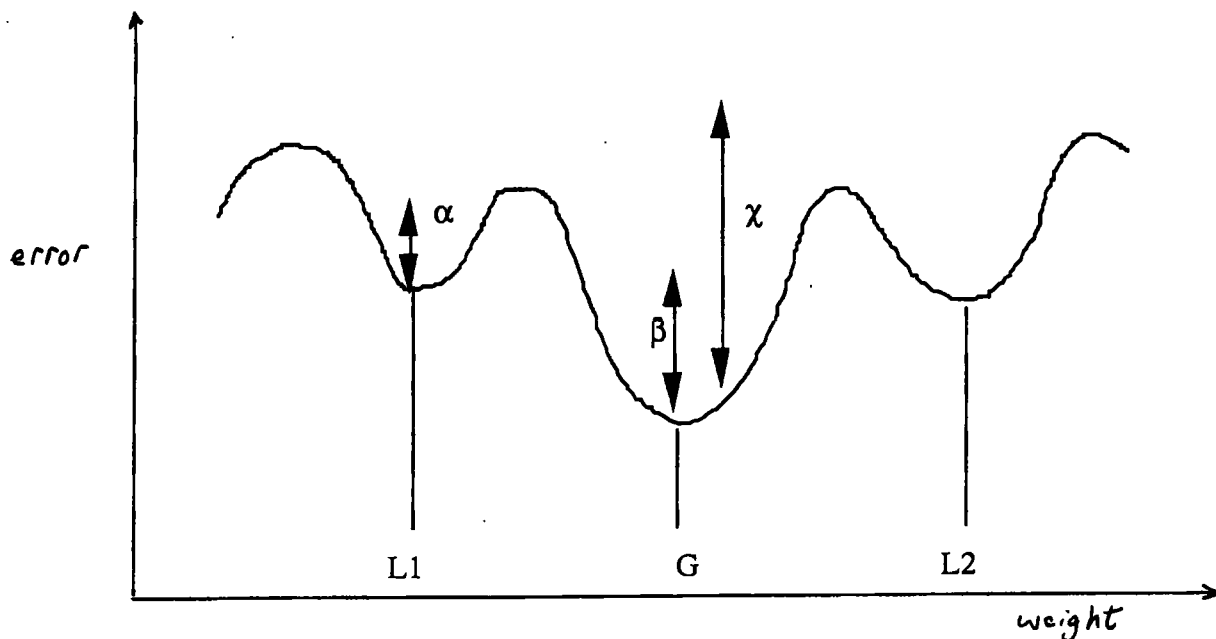
allow this latter condition to arise. The probability that the network will find a "satisfactory" minima will depend on the following three factors: i) The number of local minima ii) The number of "satisfactory" minima and iii) The complexity of the state space. The probability that a local minima will be found increases as (i) and (iii) increase and (ii) decreases.

Therefore the size of the hidden layer can adversely effect learning in one of two ways: If the state space is too small to capture the features of the data set then learning cannot take place. Does this mean that all networks should have hidden layers with large numbers of intricately connected units? The answer is no since such hidden layers bring with them large state spaces, more local minima with not necessarily more global minima<sup>3</sup>.

Another feature of the comparative study is the way in which increasing the learning rate brought with it an increase in the probability that the network would fail to converge. Of particular interest was the behaviour of the gamma model during simulations where the network had a high learning rate, where the rate of error went up as well as down. This suggests that one of two things happens when the learning rate is too large: One possibility is that the network fails to recognise the global minima when it arrives in it (see fig 3.8).

---

<sup>3</sup> This is because a global minimum in a large hidden layer is expressed in terms of a large number of connections, which means more correct connection strengths are necessary to express this and there will be more partial solutions.



*Figure 3.8: State space diagram showing the problems caused by inappropriate learning rates.*

In figure 3.8 we see a simplified state space with two local minima (L1,L2) and one global minima G. The arrowed lines represent the force exerted on the network by the learning coefficient. If the line is greater than the minima then the network will leave the minima. A network with a learning rate of  $\alpha$  will become stuck in local minima. A network with a learning rate of  $\beta$  will be able to escape the local minima but not the global minimum. Conversely a network with a learning rate of  $\chi$  will not only escape the local minima, but the global minimum as well and will continue to traverse the state space looking for a (non existent) minimum from which it cannot escape.

Another possibility is that the system is oscillating between local minima. The learning rate will be large enough to escape from local minima, but in each direction it jumps it finds itself in another local minima. Hence the network will be unable to escape from this state and will continue to oscillate.

### 3.7.2 Why Different Recurrent Networks Behave Differently

Before looking at the performance over the different networks over the different sequence processing tasks, it is worth remembering that, particularly in problems of physiological and psychological modelling, researchers will be looking for a particular pattern of behaviour as a criteria for success, rather than simply looking for the lowest possible error. Thus, in the Letter in Word Prediction Task, the sudden increase in error between words is only to be expected. This would mean that the lowest possible error rate would not be zero.

It is clear, however, that the RTRL network is the weakest of the three. This would seem to be inconsistent with the observation made by Zipser (1989) that "RTRL has been shown to have great power and generality". One possible way to resolve this apparent contradiction is the possibility that because RTRL is computationally expensive it is slower than the SRN or the Gamma network. This would be true both in terms of the number of learning epochs and real time.

The other possibilities for the poor performance of the RTRL network are concerned with deficiencies with the network. The RTRL network has a very high level of connectivity (each unit in the RTRL layer is connected to every other unit in the RTRL layer). This leads to many degrees of freedom even in a small RTRL network. Corresponding to this is the fact that the large number of modifiable connections gives a highly complex state space, which may contain many local minima. Thus during learning when the network attempts to traverse the state space it is hardly surprising that the network takes a long time to traverse it or fails to learn the task at all. Corresponding to a complex state space is a complex computationally expensive learning rule, where one weight change is a function of all other nodes in the network. Not only does this increase the amount of time that the network needs to cycle a given number of epochs, it also appears that the algorithm is unable to deal with the complex state space that the network has to traverse.



It is interesting to note that in a paper which gave details of successful training of an RTRL network (Catfolis 1993) the network size was small, the learning rate was small and a modified form of the RTRL learning rule was used (see chapter four). In this study we similarly see that the network performs best when its architecture is small and the task is well defined. When performing the XOR with two step delay task the RTRL layer was smaller than for any of the conditions tried for the other three tasks, yet performance was better than for any of the other three tasks.

The results seem to confirm observations made by Tsoi and Back (1995) that the RTRL algorithm is the least satisfactory recurrent algorithm. In Tsoi and Back's studies the RTRL network was given a longer training period than the other networks tested, but still provided the poorest results. Furthermore architectures which employ local as opposed to global feedback connections are advocated as being superior. The Locally Recurrent Globally Feed forward (LRGF) class of recurrent networks, of which the gamma model is one example is capable of providing a rich architecture which is able to solve non trivial problems.

Overall the statistical analysis of the results showed that the Simple Recurrent Network was the most effective network, narrowly outperforming both gamma models and easily outstripping the RTRL network. One possibility is that the tests used were biased towards the Simple Recurrent Network since two of the four tests used (Elman's letter in word prediction task and the finite state grammar) are drawn from papers which take the Simple Recurrent Network as their subject. There was no significant difference between the Simple Recurrent Network and the two gamma models on the dollar to Swiss Franc exchange rate problem and the Simple Recurrent Network was the worst network over the XOR with two step delay problem, which was drawn from Zipser's paper on the RTRL algorithm.

With regard to the gamma model, Table 3.12 shows a number of cases where  $\mu=1$  or takes a value very close to it. The gamma model is a tapped delay line when  $\mu=1$ . The results also showed that the gamma memory parameter  $\mu$  decreased in size

as problem complexity increased. This is because, in order to solve complex sequential problems, a recurrent network will have to remember events that have taken place further back in the past than when it is processing more simple problems.

It is particularly interesting to compare the respective values of  $\mu$  and the memory depth evolved of the different sized gamma kernels when they were tested on the XOR with two step delay task. The gamma network with a kernel size of one performs less well than the gamma network with a kernel size of two. The memory depth evolved by the latter (1.919) is of course very close to the time interval in the data (two time steps).

It is also clear that, as problems become more complex, the size of gamma kernel used does not have to increase in size in proportion to this complexity. Indeed, on the Dollar - Swiss Franc exchange rate data set, there was no statistically significant difference between the model with a kernel size of one and the model with a kernel size of two, although given the general ineffectiveness of both models not too much should be read into this result.

Another feature of the size of  $\mu$  in relation to problem complexity is that the range of the values of the  $\mu_i$  obtained over a number of trials increases as problem complexity increases. This could be for one of two possible reasons: either a wider range of values for  $\mu$  are needed if complex problems are to be solved or the network is unable to find a solution. In the latter case a wider range of values for  $\mu$  is simply a reflection of a fruitless search (e.g. as may possibly be the case for the Dollar / Swiss Franc problem). This may well be a reflection of the inability of SRN, RTRL or the Gamma model to predict future exchange rates from past exchange rates alone, presumably because exchange rates may not be predictable at all from their past.

Useful results from financial data prediction problems have been obtained. Using a Simple Recurrent Network, McCann and Kalman (see section 1.3 above) created a trend predictor for the gold bullion market. Their study claimed that "useful predictions can be made without the use of more extensive market data or knowledge". This would suggest that the networks in our comparative study which

performed poorly on the financial data might well do so if their internal parameters allowed them to better model the dynamics of the exchange rate system. It should also be noted that McCann and Kalman did not look at currency markets, but at the gold market. It may well be the case that different factors affects these two areas and that the factors which influence movements in the price of gold are easier to model than those which produce exchange rate fluctuations. Furthermore McCann and Kalman tried to predict turning points (i.e. when the price started to rise or fall) which may be easier than predicting an exchange rate from one moment to the next.

One interesting feature is that the Simple Recurrent Network is sometimes more effective at learning the data described above. This seems to be something of a paradox when one considers the respective learning rules of the Simple Recurrent Network and the RTRL networks: whilst the Simple Recurrent Network learning rule is a straightforward extension of the traditional simple back-propagation learning rule, the RTRL learning rule is a variation of the BPTT algorithm and would therefore seem to be better suited to sequence processing. A key factor however may be the computational expense of the RTRL learning rule and the BPTT algorithm from which it was derived.

### **3.8. What Next?**

There seems to be little reason (in terms of error scores) to choose the gamma model over the SRN in this study. Principe and Turner (1994) reported on a gamma network which included several modifications from the original gamma model. Chief amongst these modifications was that a different value of  $\mu$  was calculated for each unit within each kernel (as opposed to the simple form of the model which has one  $\mu$  parameter for all the units in each kernel). This modified gamma network outperformed a more conventional recurrent network. Principe and Turner reported that for the problem they examined (word spotting) a more coarse grained (i.e. low resolution) representation of past inputs was sufficient and indeed advantageous.

Whilst this may not be the case for other sequence processing problems, the ability of the gamma network to adjust its internal memory parameters is a clear advantage.

If we wish to improve the ability of recurrent networks to perform sequence processing tasks, we can attempt to do so in one of two ways: create a more favourable state space: one which is as simple as possible, with fewer local minima, yet is still able to capture the properties of the sequence which is to be learned.

Alternatively we could create a better learning algorithm: one which is able to find the global minima more quickly and is better able to avoid or escape local minimum. In the next chapter we shall attempt to apply these two approaches to the RTRL algorithm to try and improve its performance.

## **Chapter Four. Modifications to the RTRL Algorithm and Their Implications**

### **4.1. Overview and Rationale**

In chapter three, a comparative study of three different recurrent network models was undertaken. The results of the study confirmed what various other comparative studies have shown: that different recurrent network models have different sequence processing abilities. In the comparative study described in the previous chapter the models tested were duplications of the recurrent networks as they were originally proposed. However one feature of research in this field has been the modifications to these (and other) models that have been proposed by various researchers in order to try and improve their learning and generalisation ability.

In this chapter, two improvements to the Real Time Recurrent Learning (RTRL) network that have been proposed will be examined to see if they lead to an improvement in the poor performance (over all tasks save the XOR with two step delay) that this model demonstrated in chapter three. One of these modifications is concerned with the architecture of the network whilst the other is concerned with the learning algorithm itself. Each of these modifications will be described in turn and their performance on the data sets used in chapter three will be reported. In addition, the learning rate will also be varied when a modified form of calculating the  $P_{ijk}$  values are used, in order to see if this modified  $P_{ijk}$  calculation allows the use of higher learning rates (see section 4.3 below). For each simulation, the networks used to perform the XOR with two step delay problem have four hidden units, whereas for the other three problems the network has twenty hidden units. As in chapter three each data set was presented to the network ten times unless otherwise stated. This design was chosen because limitations on time and computing resources meant that ten trials was the most feasible number for testing to see if the networks were

sensitive to initial conditions. Obviously the results would be more statistically reliable if more trials were undertaken.

## **4.2. Modifying RTRL Network Architecture**

A significant feature of the RTRL network architecture is the high level of interconnectedness between units and the resulting large number of non-local connections. This architecture gives rise to a complex state space with many degrees of freedom, which increases the chances of the network falling into a local minimum during learning.

### **4.2.1 Pruning Network Architectures**

If the architecture of a network has to be large enough to develop an internal representation which allows it to perform a particular task, yet small enough to avoid the problem of overfitting (i.e. where the state space is too complex for the problem), how can an ideal architecture be found? One way suggested by Giles and Omlin (1994) is to use a pruning algorithm. The basic definition of a pruning algorithm is one which severs connections or deletes units within a network until the minimum architecture for performing a particular task remains.

Giles and Omlin carried out their research on a fully connected recurrent network using the RTRL algorithm. This form of recurrent network would particularly benefit from pruning because the high level of interconnectivity and the number of calculations required at each time step means that RTRL networks with large numbers of nodes are very computationally expensive and it would speed up learning considerably if the smallest possible network could be used.

The pruning algorithm is fairly simple: Start with a large network and present the training data to it. If the training is successful (i.e. the network converges within a given number of epochs) remove the neuron from the RTRL layer that has the

smallest weight vector. Then retrain the network as above. If a network with N RTRL units fails to converge, take the network with N+1 RTRL units as the network which is accepted. After the training data was learned a small number of negative examples were added to the training data.

Giles and Omlin showed that their algorithm gave rise to an improvement in network performance as the number of RTRL units were reduced, although retraining of the network became more difficult during this process. See table 4.1:

Neurons	Time	Size	NN Performance
15	197	142	6.75%
14	7	46	6.89%
13	98	99	2.61%
12	11	62	1.51%
11	14	67	0.97%
10	22	83	1.26%
9	111	157	2.95%
8	102	140	2.44%
7	104	118	0.14%

*Table 4.1: Table showing Summary of results from Giles and Omlin (1994). Results are after each pruning cycle. Summary of Table Headings: Neurons = size of network; Time = number of epochs before convergence; Size = size of maximum training set (see text); NN Performance = rate of error on test data.*

The time taken after pruning can be taken as an indication of the level of activity of the pruned neuron. A short training time can be taken to indicate that the neuron which was inactive had little involvement in developing an internal representation of the task. The fact that retraining time increases as the number of neurons decreases can be seen as a "weeding out" of these peripheral neurons during the initial stages of

pruning. However as the number of peripheral neurons decreases the probability that a pruned neuron will play a more significant role in the internal representation created by the network increases.

The problem with a pruning approach however is that it assumes that the network has a level of state space complexity equal to or greater than the task requires. Because pruning will always simplify the state space then no solution will ever be reached if the architecture fails to satisfy the above criteria. Furthermore, as was demonstrated with the experiments in learning the XOR with two step delay task in chapter four a pruning approach is particularly problematic if the network is small.

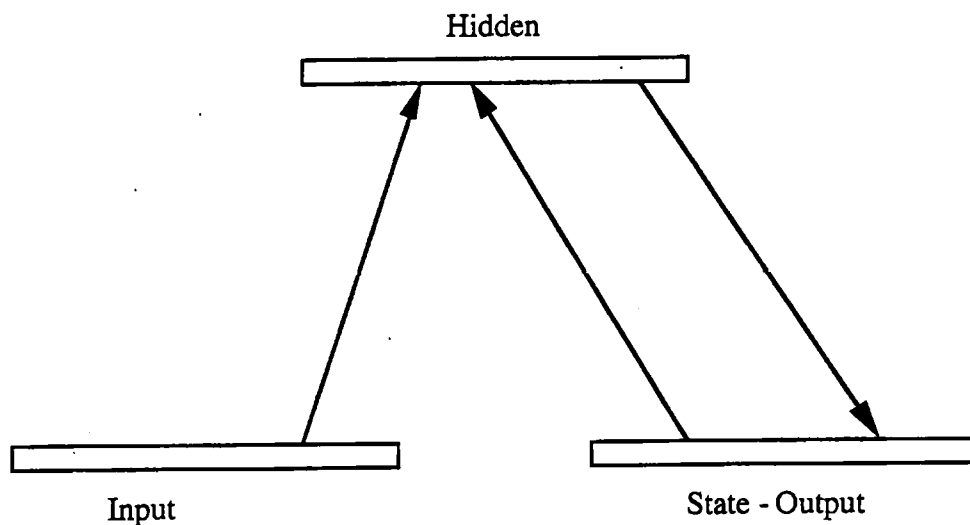
#### **4.2.2. A pruned RTRL Architecture**

One way round this problem is to reduce the number of degrees of freedom by reducing the number of connections in the network. One way to do this would be to randomly prune a number of connections. However the method we have chosen causes the network to be equivalent to the network architecture proposed by Manolios and Franelli (1994) which took a three layered feed forward network and added recurrent connections (see figure 4.1). This architecture, they argue, is the simplest universal approximator for sequence processing tasks. There are a subset of state units which are considered as output units for the purposes of learning, as in the original RTRL architecture. Using back propagation through time as the learning rule, these networks proved to be capable of learning a subset of the seven Tomita grammars (see section 5.3.2 for a full description of this data). Manolios and Franelli (1994) trained their networks on grammars one, two, four and six. The Back Propagation Through Time algorithm was used. Weight updates were done at the end of each string.

Manolios and Franelli (1994) showed that small sparsely connected recurrent networks are able to learn complex tasks. For grammars one and two the network has one input unit, two state units (one of which is designated as an output unit) and two

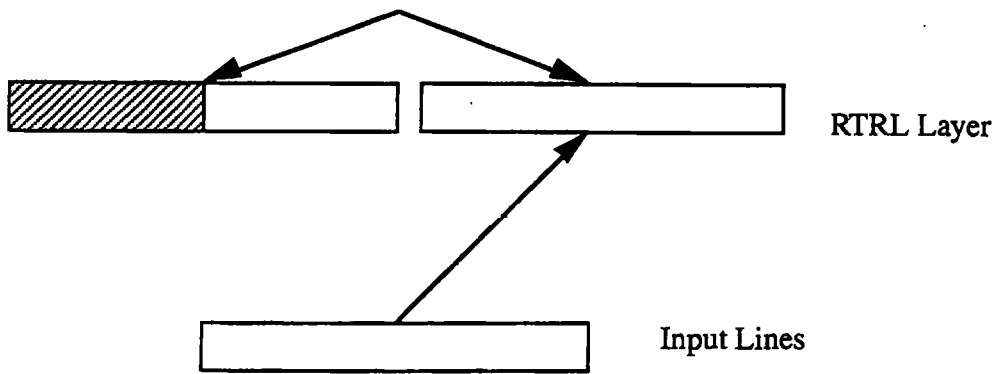


hidden units. The architecture is identical for Tomita grammars four and six except that three hidden units are used. Even on the hardest task (grammar six) the network was able to learn the data such that it could classify all the test data to within 0.02 of the desired output.



*Figure 4.1: Architecture proposed by Manolios and Franelli (1994). Arrows indicate full connectivity between layers.*

Applying the ideas behind the architecture used by Manolios and Franelli (1994) the result is that the RTRL layer is split in two. Connections are pruned so that the portion of the RTRL layer which contains the output units does not receive direct connections from the input layer. A two way connection exists between the two portions (see figure 4.2). Nodes within the same portion do not connect to each other. Running this network required the writing of a modified form of the RTRL learning rule, details of which can be found in Appendix 4. Note that this rewrite was purely in terms of getting the learning rule to work with a sparsely connected architecture, not because the underlying formulae were different in any way.



*Figure 4.2: A Sparse RTRL network architecture of the type described by Manolios and Franelli (1994). Arrowed lines indicate full interconnectivity. The subset of output units are shaded. Both portions of the RTRL layer are fully interconnected to each other. As indicated earlier, nodes within the same portion are not connected to each other.*

This network is trained with the traditional RTRL algorithm. However the reduced number of connections leads to the calculation of the  $P_{ijk}$  variable being less computationally expensive, as well as decreasing the complexity of the state space which the network traverses during learning. This sparse architecture was tested against the four sequence processing tasks described in chapter three. Details of each architecture are shown in table 4.2.

	Input Layer	Portion 1	Portion 2	Output Subset
2 delay XOR	2	2	2	1
Letter in word prediction	5	10	10	5
Finite state grammar	3	10	10	3
Exchange rate prediction	10	10	10	10

*Table 4.2: Details of architectures used for the modified RTRL architecture. Portion 1 and portion 2 refer to the divided RTRL layer. Only portion 1 receives input from the outside world. The output subset lies in portion 2. Table entries are numbers of units.*

Each of the architectures was tested ten times to evaluate the effect of initial conditions on learning. For all simulations the learning rate was fixed at one. Results are shown in table 4.3.

	Original RTRL Model	Sparse RTRL Model
2-delay XOR	0.240	0.180
Letter in Word Prediction	1.011	1.002
Finite State Grammar	0.816	0.925
Dollar - Swiss Franc Exchange Rate	1.129	1.184

*Table 4.3: RMS Error scores showing the performance of the sparse RTRL architecture described above against the traditional RTRL architecture advocated by Williams and Zipser.*

As with the work done in chapter three, these results were then subjected to a more thorough statistical analysis. Table 4.4 shows a comparison over the XOR with two step delay task, Table 4.5 shows a comparison over the Elman letter in word prediction task, Table 4.6 shows a comparison over the finite state grammar. Finally table 4.7 shows a comparison over the Dollar / Swiss franc exchange rate data.

	Original RTRL Model	Sparse RTRL Model
Mean	0.240	0.180
Standard Deviation	0.055	0.013
Range: Min	0.219	0.153
Max	0.403	0.199
Confidence Limits	0.240±0.013	0.180±0.003

Table 4.4 Statistical analysis of the <sup>error</sup>results of the original and sparse RTRL architectures over the XOR with two step delay task.

Analysis of variance showed that there was a significant difference between the two networks (p<0.01), with the sparse RTRL outperforming the original fully connected model.

	Original RTRL Model	Sparse RTRL Model
Mean	1.011	1.002
Standard Deviation	0.095	0.063
Range: Min	0.857	0.953
Max	1.215	1.080
Confidence Limits	1.011±0.021	1.002±0.014

Table 4.5 Statistical analysis of the results of the original and sparse RTRL architectures over the Elman letter in word prediction task.

Unlike the XOR with two step delay, analysis of variance showed that there were no significant differences between the original and sparse RTRL models on this task (*Elman letter-in-word*).

	Original RTRL Model	Sparse RTRL Model
Mean	0.816	0.925
Standard Deviation	0.009	0.055
Range: Min	0.802	0.879
Max	0.828	0.997
Confidence Limits	0.816 $\pm$ 0.002	0.925 $\pm$ 0.014

*Table 4.6 Statistical analysis of the results of the original and sparse RTRL architectures over the finite state grammar task.*

Analysis of variance shows that the original RTRL network outperformed the sparse RTRL model ( $p < 0.001$ ) for the finite state grammar task.

	Original RTRL Model	Sparse RTRL Model
Mean	1.129	1.184
Standard Deviation	0.045	0.084
Range: Min	1.095	1.108
Max	1.247	1.314
Confidence Limits	1.129 $\pm$ 0.011	1.184 $\pm$ 0.019

*Table 4.7 Statistical analysis of the results of the original and sparse RTRL architectures over the Dollar /Swiss Franc exchange rate task..*

Analysis of variance showed that there were no significant differences between the original and sparse RTRL models on this task (*exchange rate*).

The above analysis shows that the sparse RTRL model does not do significantly better than the original fully connected RTRL model apart from the XOR with two step delay task, a task which Zipser (1990) describes as being "too simple a problem to provide a meaningful test" (Zipser 1989 pp 556). On this data, a sparsely connected RTRL model does little to bridge the gap between it and other recurrent network models such as the simple recurrent network and the gamma model.

Another way to compare these two versions of RTRL is to examine the time taken for the networks to converge to a solution. Although of course the most important property of a neural network is its ability to find a solution, it is also desirable that such a solution should be found as quickly as possible. Figures showing the convergence performance of the original and sparse RTRL architectures for the XOR with two step delay are shown in table 4.8. In order to arrive at these figures each network was run ten times. Each network had two inputs and four RTRL units (one of which was an output unit). The learning rate was set to one.

	Original RTRL Model	Sparse RTRL Model
Maximum	8599	11715
Minimum	5902	5142
Mean	7395.8	7596.5

*Table 4.8 The Maximum, minimum and mean number of trials needed for the original and the sparse RTRL Model to converge for the XOR with two step delay task.*

These figures would seem to indicate that although there is little difference in mean convergence times, the sparse RTRL architecture seems to have a wider range of convergence times than the fully connected RTRL architecture. This may reflect the fact that the state space of the sparse RTRL architecture is slightly more sensitive to initial conditions than the fully connected RTRL architecture, though not so much as to effect the networks ability to learn or the time it takes to converge significantly.

Why does the sparse RTRL architecture perform differently to the original RTRL architecture? For a particular task, a network needs to have a mechanism which is sufficiently complex to capture the properties of what it is trying to model whilst not being so complex as to face excessive numbers of local minima in which the network may become trapped during learning. If the sparse RTRL architecture outperforms the original RTRL architecture, it may well be the case that the original RTRL architecture is too complex for this particular task. This is not to say that the original RTRL architecture is incapable of learning the task, rather that a less complex architecture (i.e. one with fewer connections or units) could do the job equally well. Conversely, if the original RTRL architecture outperforms the sparse RTRL architecture, it will be the case that the sparse RTRL architecture is not sufficiently complex (i.e. the network has insufficient connections or units) to perform the task in question.

The work of Manolios and Franelli shows that sparsely connected small recurrent networks are capable of learning complex sequence processing tasks. This may partly be due to the fact that the overall set of training strings is small and the length of each individual string is quite short (no string had a length greater than four). This suggests that whilst fully connected architectures are able to deal with problems which require a relatively short term memory mechanism they have difficulty in dealing with problems with longer strings or where the length of string could vary widely (i.e. all the tasks used in our comparative study except for the two step delay XOR). The original network proposed by Manolios and Franelli would also cause difficulties over these data sets because of the computational expense of the Back Propagation Through Time algorithm (a copy of the network is needed for each time step).

The failure of the modified RTRL architecture to match the performance of either the Simple Recurrent Network or the Gamma Model suggests two possible alternatives: Either the non local nature of the RTRL algorithm is the main obstacle to learning and no tampering with the architecture will get round this, or sparse

connectivity can solve the problem, but the network needs to be sparse in a different way. If the second point is to be proven then there needs to be an exhaustive check of all the various sparsely connected architectures that exist. Only when this is done can the first point be accepted or rejected. Since this would be a long drawn out process, it is worth noting that researchers have attempted to automate the process of finding the optimum architecture for a given task (see section 4.2.1 and section 6.2).

### **4.2.3 Randomly Pruning Connections During Learning**

The work of Manolios and Franelli (1994) uses an architecture which has been pruned before learning takes place. However there is an alternative way to reduce the level of connectivity in the network. This alternative examines the connections between units at pre-set intervals and removes those connections according to some criteria. An example of this type of algorithm was used by Giles and Omlin (1994) (see section 4.2.1 for further details). In this section we report on the findings of the use of a pruning strategy on an RTRL network attempting to learn the finite state machine used in Chapter Three. The pruning criteria was as follows: at a given point in the learning schedule, a given percentage of connections of smallest absolute magnitude are to be disabled. Three different values of pruning points (once per presentation of training data, twice per presentation of training data and once every two presentations of training data) and percentage of weights to be pruned (1%, 5% and 10%). This creates nine separate conditions.

For each of the above conditions training consisted of ten presentations of the training data (thus for the above conditions pruning occurred ten, twenty and five times respectively). The learning rate in all conditions was one. The network consisted of three input units and twenty RTRL units, three of which served as output units. The performance of the networks when trying to predict the test data is shown in table 4.9:



	Prune once every two presentations	Prune once every presentation	Prune twice every presentation
Prune 1%	0.655	0.66	0.657
Prune 5%	0.663	0.875	0.882
Prune 10%	0.716	0.788	0.834

*Table 4.9 Root Mean Squared Error of Test data for Finite State Machine learning task using an RTRL network and a pruning algorithm*

Note that the root mean squared error score for the original RTRL algorithm over this task was 0.665. On this evidence it would seem that for the best results, pruning should not take place very often and the percentage of connections pruned should be small. However none of the combinations of interval between pruning or percentage of connections pruned proved capable of significantly improving the performance of the RTRL network. Indeed some combinations only lead to inferior performance. The probability of such a deterioration seems to increase when the interval between pruning is short and the percentage of connections pruned increases.

As with the pruned architecture discussed in section 4.2.2, this method was also tested with a view to examining the time taken for the networks to converge to a solution. For this the different pruning variables were tested on the XOR with two step delay task. All other variables concerning the network were constant: each network had two inputs and four RTRL units (one of which was an output unit). The learning rate was set to one. The results are shown in table 4.10.

	Prune once every 2000 patterns	Prune once every 1000 patterns	Prune twice every 500 patterns
Prune 1%	Max= 10803 Min= 5334 Mean= 7192	Max= 9630 Min= 5896 Mean= 8209.375	Max= 10094 Min= 5672 Mean= 7485.2
Prune 5%	NL	NL	NL
Prune 10%	NL	NL	NL

*Table 4.10: Maximum (Max), minimum (Min) and Mean number of trials to convergence for different pruning variables, tested on the XOR with two step delay task. Note that NL means that the network did not converge on any of the ten trials.*

The results summarised in table 4.10 would seem to suggest that the network performed best when pruning took place relatively infrequently and the number of connections pruned was small. Even so none of the mean convergence times were significantly lower the mean convergence time for an unmodified RTRL architecture which were 8599, 5902 and 7395.8 for maximum, minimum and mean convergence respectively.

Table 4.10 also reveals the difficulties that can result from too much pruning. If too many connections are pruned then the network loses the ability to learn completely. This is true both in terms of the interval between pruning being too short or if the percentage of connections pruned is too high.

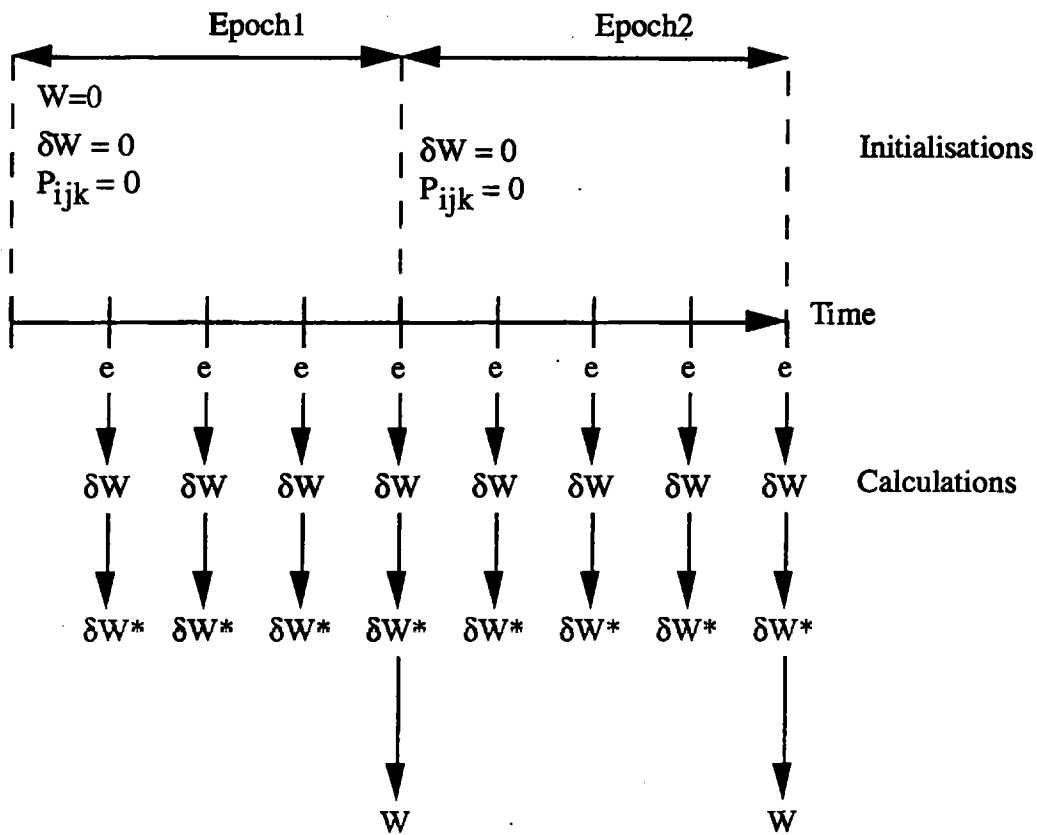
The instability caused by too much pruning becomes even more apparent when one considers the fact that under only one set of conditions (prune 1% of connections once every 2000 patterns) did the network converge every time. When 1% of connections were pruned once every 1000 patterns the network failed to converge on two occasions. When 1% of connections were pruned once every 500 patterns the network failed to converge on five occasions.

### 4.3 Modifying the RTRL Algorithm

The architecture of an RTRL network has a large number of non local connections. These are used by the RTRL learning rule in that a change in one weight is a function of all the other weights in the network. This algorithm is computationally expensive, particularly when the number of RTRL units is large. But according to Williams and Zipser (1989) this algorithm has the benefits of great power and generality, which, if it were true, would be well worth the computational expense.

One way to preserve this power and generality in the face of the problems of local minima described in section 4.2 is to zero the  $P_{ijk}$  variable after a set number of input patterns have been presented to the network. If the interval is set to a value of five for example,  $P_{ijk}$  is calculated in the normal way for the first four inputs and is set to zero on the fifth input. The process is then repeated until the desired number of pattern presentations have been reached. Note that whilst  $P_{ijk}$  values may be lost as the result of this resetting, the weight changes are preserved.

This method was first proposed by Catfolis (1993). Schematics of the original RTRL algorithm proposed by Williams and Zipser and the modification proposed by Catfolis are shown in figures 4.3 and 4.4. In the schematic of the Catfolis version of RTRL,  $P_{ijk}$  is reset to zero after  $\tau$  presentations to the network. The effect of periodic resetting of  $P_{ijk}$  values is to "jolt" the network out of local minima. This works because resetting  $P_{ijk}$  causes a significantly different modification to connection strengths within the network than would otherwise be the case. The idea of somehow restricting the network during learning is one which has been discussed elsewhere in the literature, the detailed mechanics of this process is discussed in section 5.4. and by Elman (1993).



*Figure 4.3 The Original form of calculating the  $P_{ijk}$  portion of the RTRL algorithm proposed by Williams and Zipser (1989).*

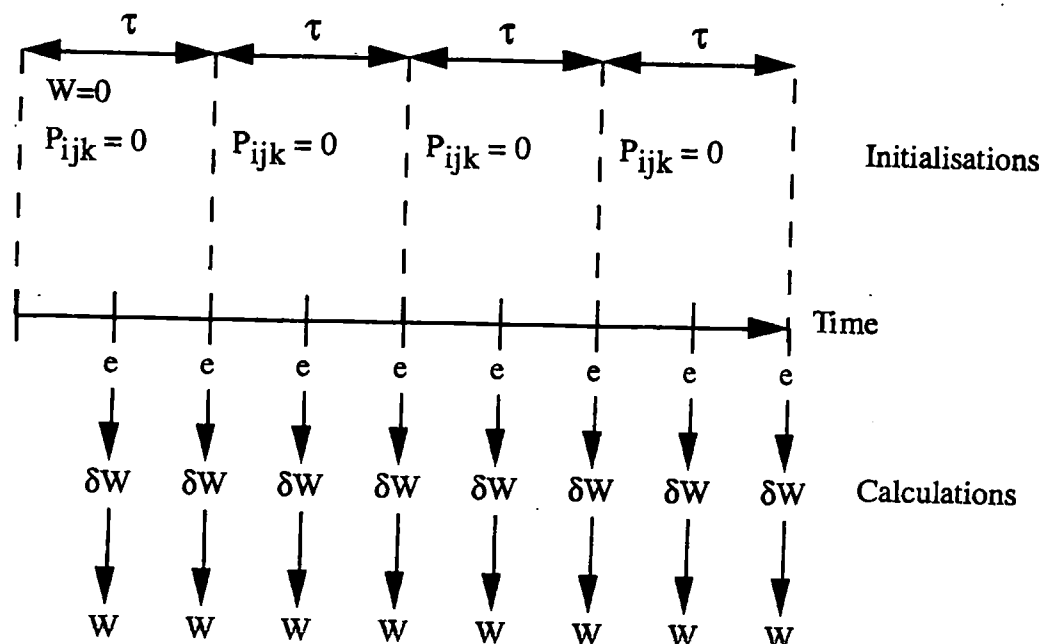


Figure 4.4 A Modified form of calculating the  $P_{ijk}$  portion of the RTRL algorithm proposed by Catfolis (1993).

Does this method of avoiding local minima lead to an improvement in learning? Comparative results of the original and modified RTRL algorithms are shown in table 4.11. In all of the tests described below the reset interval  $\tau$  for the modified RTRL algorithm is set to four. In all other respects the two networks are identical.

	Original RTRL Algorithm	Modified RTRL Algorithm
2-delay XOR	0.240	0.123
Letter in Word Prediction	1.011	0.903
Finite State Grammar	0.816	0.840
Dollar - Swiss Franc Exchange Rate	1.129	1.114

Table 4.11 Comparative results of the original and modified RTRL algorithms (RMS).

As in section 4.1, these figures were subjected to a more detailed statistical analysis.

	Original RTRL Algorithm	Modified RTRL Algorithm
Mean RMS error	0.240	0.123
Standard Deviation	0.055	0.032
Range: Min	0.219	0.085
Max	0.403	0.183
95% Confidence Limits	0.240±0.039	0.123±0.023

*Table 4.12 Statistical analysis of the original and modified RTRL learning algorithm over the two step delay XOR task.*

An analysis of variance on the above results showed that the RTRL network with the modified learning algorithm did significantly better than the original RTRL algorithm ( $p < 0.01$ ).

	Original RTRL Algorithm	Modified RTRL Algorithm
Mean RMS error	1.011	0.903
Standard Deviation	0.095	0.021
Range: Min	0.857	0.883
Max	1.215	0.940
95% Confidence Limits	1.011±0.021	0.903±0.005

*Table 4.13: Statistical analysis of the original and modified RTRL learning algorithm over the Elman letter in word prediction task.*

As in the case of the XOR with two step delay task, the modified version of the RTRL algorithm performed significantly better than the original RTRL algorithm ( $p<0.01$ ).

	Original RTRL Algorithm	Modified RTRL Algorithm
Mean RMS error	0.816	0.840
Standard Deviation	0.003	0.006
Range: Min	0.802	0.832
Max	0.828	0.849
95% Confidence Limits	$0.816\pm0.002$	$0.840\pm0.001$

*Table 4.14: Statistical analysis of the original and modified RTRL learning algorithm over finite state grammar task.*

In this case analysis of variance showed that there was no significant difference between the two forms of the RTRL algorithm.

	Original RTRL Algorithm	Modified RTRL Algorithm
Mean RMS error	1.129	1.114
Standard Deviation	0.044	0.010
Range: Min	1.095	1.099
Max	1.247	1.133
95% Confidence Limits	$1.129\pm0.011$	$1.114\pm0.002$

*Table 4.15: Statistical analysis of the original and modified RTRL learning algorithm over the Dollar to Swiss Franc exchange rate prediction task.*

As was the case with the finite state grammar task, there was no significant differences between the original and modified RTRL algorithms over this data set.

Thus modification of the learning algorithm would seem to lead to a greater improvement than modification of the architecture, giving rise to significant improvements over two tasks (XOR with two step delay and Elman's letter in word prediction task).

#### **4.3.1 Different Reset intervals for different tasks?**

One additional factor to be considered when examining the learning capabilities of the modified RTRL algorithm is that the interval between resetting the  $P_{ijk}$  variable is of critical importance: too short an interval renders the short term memory mechanism ineffective, since the network will not be able to hold all the information it needs to perform a particular task. Conversely too long an interval causes the network to more closely resemble the original RTRL model and any advantage that might be gained by using this method is lost. Thus there exists a range of reset values which will affect learning in some way. What is more, since different tasks require the network to process sequences of varying length, this critical range will differ for different tasks.

The effect of different  $\tau$  values on the learning ability of the network is also supported by Catfolis (1993), who concluded that the  $\tau$  value needs to be as close as possible to the temporal requirement of the problem. If the  $\tau$  value is significantly higher than the temporal requirement of the problem, then the network will receive all the information it needs "but the weights will change too much. The direction of the weight change will not follow the true gradient of the total error" (Catfolis 1993 pp815). Conversely, if the  $\tau$  value is significantly lower than the temporal requirement of the problem, the network will not receive all the information it needs and generalisation will be poor.

An illustration of this can be found when we consider the four data sets used in this comparative study. The XOR with two step delay problem requires the network to recall inputs from a fixed point in the past (i.e. two time steps



previously). In contrast the letter in word prediction and finite state grammar tasks require the network to process sequences of differing lengths. Prediction of the Dollar / Swiss Franc exchange rate is more complex still: each market movement being a combination of short and long term factors.

An example of this can be found when training the network on the XOR task with two step delay. If the  $\tau$  value is set to three the network performs less well than the original RTRL algorithm over ten presentations of the data set. If however the network is given another ten presentations of the data set then the RMS error score over the test data is as good as the original RTRL algorithm. Indeed performance is slightly improved after twenty presentations. This is shown in table 4.16.

Original RTRL algorithm after ten presentations of data	Modified RTRL after ten presentations of data	Modified RTRL after twenty presentations of data
0.240	0.501	0.144

*Table 4.16 Comparative RMS error scores over the XOR with two step delay task between the original RTRL algorithm and the modified RTRL algorithm. when the  $P_{ijk}$  value is set to zero after every three presentations.*

One possible explanation for this is that resetting  $P_{ijk}$  values at such a relatively short interval hampers the network in its attempts to converge, but not sufficiently as to stop the learning process altogether. Resetting  $P_{ijk}$  after every three presentations may well reset  $P_{ijk}$  before the information held in memory can be used to correctly solve the problem at a particular time step. Therefore the collection of "uncut" strings will take longer to build up, resulting in slower convergence. Interestingly this problem requires a memory of fixed length, since the desired output at time  $t$  is always the solution to the input at time  $t-2$ , whilst the interval between resets is larger than the required memory length. This suggests that  $P_{ijk}$  values from

one string of data<sup>1</sup> can assist in the learning of another string of data. Of course if the strings are of variable length resetting of the networks'  $P_{ijk}$  values will take place in the middle of data strings as well as at their beginning or end. This handicap may be overcome by increasing the learning rate (see section 4.3.2). Another factor to take into consideration is that XOR with two step delay is a continuous problem, unlike the finite state grammar or the Elman letter in word prediction task.

The effect of reset interval on network performance can also be seen when examining the results of a pattern set on which RTRL did less well such as the finite state grammar, as shown by the results in table 4.17

Original RTRL Algorithm	Reset =2	Reset =3	Reset =4	Reset =5
0.665	0.738	0.667	0.705	0.657

*Table 4.17 Comparative RMS error scores over the finite state grammar task between the original RTRL algorithm and the modified RTRL algorithm with different reset values.*

The results in table 4.17 seem to show that the error scores fluctuate, rather than a smooth decrease followed by an increase as one moves through the range of critical reset values. Accordingly the search for an optimal  $\tau$  value must be fairly exhaustive. It may not always be enough to start with one value and increment by one so long as the mean RMS error across the test data is lower than the previous reset value. Although the search space for the optimal  $\tau$  value is fairly rich, the cost is that finding the optimal value may well be a long drawn out process.

---

<sup>1</sup>In this context the term "string of data" or "data string" refers to a grammatical string of inputs and desired outputs within a pattern set, such as whole word for the letter in word prediction task or one complete traversal of the finite state grammar.

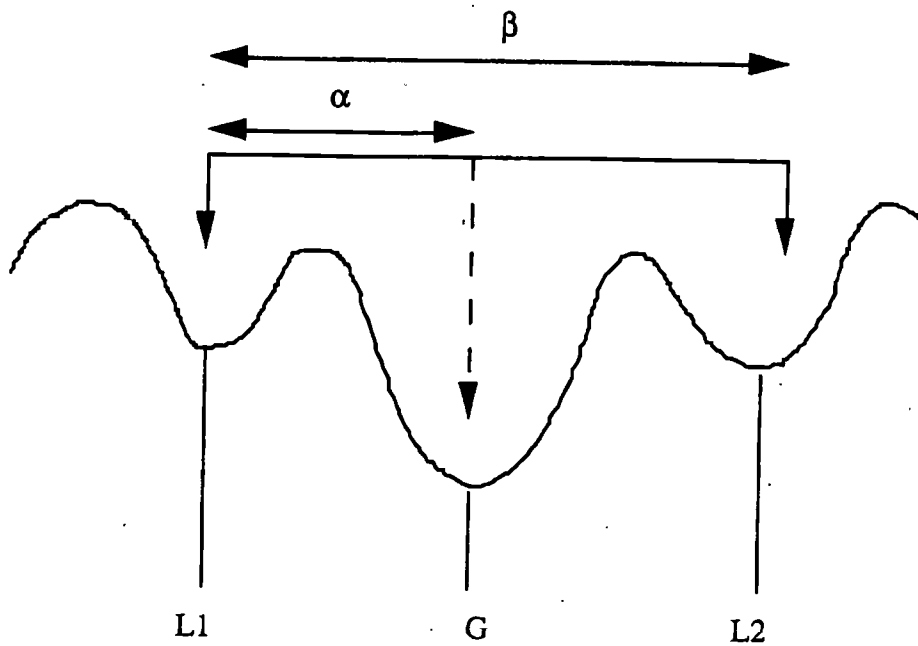
Are there any heuristics that could be used by researchers to find the optimal  $\tau$  value for a particular task? Obviously the nature of the sequence processing task is of considerable importance. Catfolis (1993) suggests that the optimal  $\tau$  value is a function of the number of RTRL units:

"When the number of RTRL nodes is small (i.e. the net has a low memory capacity), the best nets will be those which are able to extract the most information from a minimal amount of time. Nets trained with a  $[\tau]$  that is related to that (small) piece of time will show the best results. The smaller the number of nodes the smaller the optimal  $\tau$  value will need to be" (Catfolis 1993 pp 816).

However Catfolis goes on to suggest that there is not an optimal  $\tau$  value, but rather there is a range of values which give better results. This supports the interpretation made in connection with the results shown in table 4.7, which does not support the idea that there exists an optimum reset value with less ideal reset values either side. Indeed setting a reset value of  $r$  as opposed to  $r+1$  can often make a significant difference. Catfolis demonstrates that zeroing  $P_{ijk}$  at different places in the training epoch means that the network gains a richer information sample, which facilitates better learning.

#### **4.3.2 Does Resetting $P_{ijk}$ Allow Increased Learning Rates?**

On the face of it, when training a neural network large learning rates would always seem to be a good thing, since the larger the learning rate the faster the network learns a task. However the reality is much more complex.



*Figure 4.5: A simplified state space showing the problems that too high a learning rate can cause.*

The problem is illustrated in figure 4.5. The diagram depicts a state space where the global minimum  $G$  is surrounded by local minima  $(L1, L2)^2$ . The state of the network is such that whilst the network lies somewhere within  $L1$ , the learning rule  $\beta$  is large enough to escape it, moving the network in the direction of  $G$ . Unfortunately the movement is large enough to miss  $G$  completely and land somewhere in  $L2$ . On the next training cycle the network moves in the direction of  $G$ , but only succeeds in landing back in  $L1$ . However with a smaller learning rate such as  $\alpha$ , the network is able to escape  $L1$  and lands somewhere within  $G$ .

---

<sup>2</sup>Given the complexity of the real state space that an RTRL network would have to traverse, it could be argued that this simplification is not a realistic situation. However since parts of the regions around a global minimum would represent some of the connection strengths found in the global minimum but not others (i.e. they would be local minima) this simplified example seems to be justified.

Because the act of resetting  $P_{ijk}$  values allows the network to escape local minima (by giving the system a "jolt"), one possibility of using this method is that we can increase the learning rate of the network without increasing the probability that the network will fail to learn at all, thus increasing the speed at which the network learns a particular pattern set. The effect of increasing the learning rate on the network is shown in table 4.18. In the experiments described below, the  $P_{ijk}$  values are reset after every four pattern presentations.

	Original RTRL Algorithm	Reset RTRL K=1	Reset RTRL K=4
2-delay XOR	0.240	0.123	0.121
Letter in Word Prediction	1.011	0.903	1.098
Finite State Grammar	0.816	0.840	0.833
Dollar - Swiss Franc Exchange Rate	1.129	1.114	1.231

*Table 4.18 RMS Error for different RTRL networks, showing the effect of increasing the learning rate (K). For all reset RTRL networks the reset interval was set to four, apart from the 2-delay when K=4, where the reset interval was set to three.*

Again, a more detailed statistical analysis of these results was carried out. The results are shown in the following tables:

Table 4.19 gives an analysis for the XOR with two step delay task.

Table 4.20 gives an analysis for the Elman letter in word prediction task.

Table 4.21 gives an analysis for the finite state grammar task.

Table 4.22 gives an analysis for the Dollar to Swiss Franc exchange rate task.

	Original RTRL	Reset L=1	Reset L=4
Mean	0.240	0.123	0.121
Standard Deviation	0.055	0.032	0.005
Range: Min	0.219	0.085	0.112
Max	0.403	0.183	0.131
95% Confidence Limits	0.240±0.393	0.123±0.008	0.121±0.001

*Figure 4.19 Statistical analysis of the original RTRL algorithm together with the modified RTRL algorithm with a learning rate of one and a modified RTRL algorithm with a learning rate of four over the XOR with two step delay task.*

For XOR, Analysis of variance showed that increasing the learning rate did not effect the performance of the modified RTRL algorithm. Both significantly outperformed the original RTRL algorithm ( $p<0.01$ ) but there was no significant difference between either of the two modified algorithms.

	Original RTRL	Reset L=1	Reset L=4
Mean	1.011	0.903	1.098
Standard Deviation	0.095	0.021	0.045
Range: Min	0.857	0.883	1.033
Max	1.215	0.940	1.189
95% Confidence Limits	1.011±0.021	0.903±0.005	1.098±0.009

*Figure 4.20 Statistical analysis of the original RTRL algorithm together with the modified RTRL algorithm with a learning rate of one and a modified RTRL algorithm with a learning rate of four over the Elman letter in word prediction task.*

Analysis of variance showed that over this task increasing the learning rate led to a significantly worse performance for the modified RTRL algorithm than both the other two networks ( $p < 0.01$ ) (*letter-in-word*).

	Original RTRL	Reset L=1	Reset L=4
Mean	0.816	0.840	0.833
Standard Deviation	0.009	0.006	0.045
Range: Min	0.802	0.832	0.744
Max	0.828	0.849	0.934
95% Confidence Limits	$0.816 \pm 0.001$	$0.840 \pm 0.001$	$0.833 \pm 0.010$

*Figure 4.21 Statistical analysis of the original RTRL algorithm together with the modified RTRL algorithm with a learning rate of one and a modified RTRL algorithm with a learning rate of four over the finite state grammar task.*

Analysis of variance showed that increasing the learning rate of the modified RTRL algorithm did not lead to a significant improvement in overall performance. Although it did bring about a greater variability in performance, as indicated by a larger standard deviation than either of the other two networks (*finite state grammar*).

	Original RTRL	Reset L=1	Reset L=4
Mean	1.129	1.114	1.231
Standard Deviation	0.044	0.010	0.063
Range: Min	1.095	1.099	1.166
Max	1.247	1.133	1.327
95% Confidence Limits	$1.129 \pm 0.011$	$1.114 \pm 0.002$	$1.231 \pm 0.014$

*Figure 4.22: Statistical analysis of the original RTRL algorithm together with the modified RTRL algorithm with a learning rate of one and a modified RTRL algorithm with a learning rate of four over the Dollar to Swiss Franc exchange rate task.*

*For the exchange rate task,*

Analysis of variance showed that increasing the learning rate brought about a significantly worse performance than either the original or modified RTRL networks with a smaller learning rate.

The most dramatic improvement brought about by an increase in learning rate is when the network is faced with the two step delay problem. The rate of error is lower than the original RTRL score whilst the reset interval used caused a slower convergence time when the learning rate was set to one (see table 4.18). The reason for this improvement may well be that the more coarse grained movement round the state space which occurs when the learning rate is increased is better able to take advantage of the "jolts" which occur when the  $P_{ijk}$  values are reset. When the learning rate was set to four, the network did not always succeed in learning the data and consequently performed poorly on the test data, whereas when the learning rate was set to one the network always succeeded in learning the data. However in the former case the network failed to learn the data on one occasion only.

As with the RTRL model with a modified architecture, the improvements that were reported were still not sufficient to give the RTRL network the same ability as the Simple Recurrent Network or the Gamma model.

The other reason for increasing the learning rate is to reduce the time that the network takes to learn the data. In order to investigate this property the performance of the RTRL network over the XOR with two step delay task was examined. The following parameters were used: Learning rate had a constant value of either one, two or four. The  $P_{ijk}$  reset value was either three, four, five or not at all. Each combination was repeated ten times and the number of presentations required for convergence was noted. The results are shown in table 4.23. Each network had an identical architecture of two input units and four RTRL units (one of which served as the output unit).



	None	$\tau=3$	$\tau=4$	$\tau=5$
L=1	7203.3	12567.5	9965.8	8877.5
L=2	3507.7	6494.8	5401.5	5242.6
L=4	2328.3*	3296.5	3312.3	3312.1

*Table 4.23 Mean convergence times (in number of trials) for different combinations of reset value ( $\tau$ ) and different learning rates ( $L$ ) over the XOR with two step delay task. A \* indicates that the network did not converge over all ten trials.*

As we can see from table 4.23 increasing the learning rate does lead to a reduction of the time that the network needs to learn the task. Indeed the fastest times are achieved with the original RTRL algorithm and  $P_{ijk}$  is never reset. This does not mean however that the modified RTRL algorithm devised by Catfolis is of no use. The speed of the original RTRL algorithm with a learning rate of four is at the cost of stability, since the network failed to converge during three out of ten trials, as opposed to all other combinations, which converged every time.

Thus it would appear that resetting the  $P_{ijk}$  values serves to suppress the tendency of the network to become trapped in local minima when the learning rate is large. So prone is the RTRL algorithm to this problem that Catfolis used a learning rate of 0.00001, far smaller than any of the learning rates used in this research. This is borne out by findings which were reported in Chapter Three, where increasing the learning coefficient was shown to lead to increased instability in some cases, notably with the gamma model. However since resetting  $P_{ijk}$  values allows the network to escape local minima then the learning rate can be increased without much loss of speed of convergence.

Overall it is clear that both changing the learning rate and the reset interval has an effect on learning. Although none of the various combinations of learning rules and reset intervals tried has significantly improved the RTRL algorithm to the extent

that they perform as well as either the Simple Recurrent Network or the Gamma Model.

#### **4.4 Summary**

In this chapter we describe and replicate two attempts to improve the performance of the RTRL algorithm by modifications to the architecture and the learning rule. Overall the results are disappointing since none of the modifications suggested brought the RTRL network up to the standard of either the Simple Recurrent Network or the Gamma Model. What they do reveal however is the rich nature of the RTRL networks behaviour and that it is best suited to tasks where the memory requirement is known and the data set small.

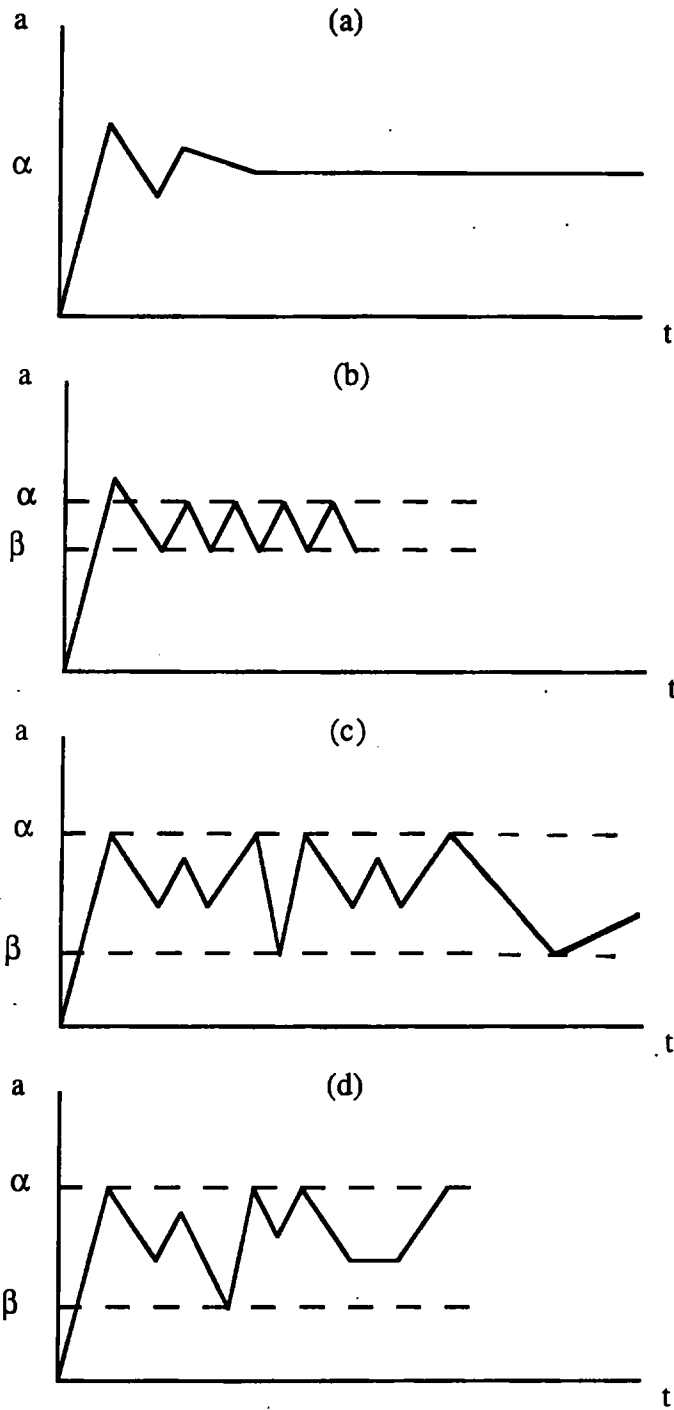
## Chapter Five. Summary and Conclusion

### 5.1 The Effect of Architecture on Learning

Of the three architectures examined in chapters three and four, the networks which have an RTRL architecture perform the most poorly, even when we consider the performance of the "improved" RTRL models discussed in chapter four. This finding is at odds with the claims of Williams and Zipser (1989) who state that the RTRL algorithm has great power and generality. One reason for this apparent contradiction is the way in which the RTRL algorithm traverses the search space. The reason for this becomes apparent when one considers the observations of Dayhoff et al (1994):

"Unlike feed-forward networks, which are static, networks with recurrent connections can exhibit periodic oscillations, quasi-periodic oscillations, and chaotic attractors as well as fixed point attractors...[Recurrent networks]...offer a wide repertoire of differing basins of attraction with complex boundaries".

Examples of these different types of attractor are shown in figure 5.1. Graph (a) represents a fixed point attractor where the activity of the system tends towards a single state  $\alpha$  over time. Graph (b) represents a periodic attractor where the activity of the system tends towards oscillating between two states  $\alpha$  and  $\beta$  over time. This idea is extended in graph (c) where activity oscillates within the region bounded by  $\alpha$  and  $\beta$  between more than two states but does so in a regular fashion. This is known as a quasi-periodic attractor. Still more complex is graph (d) which as in the case of the quasi-periodic attractor oscillates within the region bounded by  $\alpha$  and  $\beta$  but does so in an unpredictable manner. This is known as a chaotic attractor.



*Figure 5.1: Four different kinds of attractor (a) a fixed point attractor, (b) a periodic attractor, (c) a quasi - periodic attractor, (d) a chaotic attractor.*

Note that in a neural network, whilst the state space is defined by the number of connections, the attractors in this system are defined by the desired outputs of this

network since the global minimum will represent the matrix of connection strengths which produce the desired outputs of a data set.

Dayhoff et al (1994) were concerned with the ability of recurrent networks to form attractors under different conditions (size of network, different learning rates etc.) and the robustness of this ability to perturbations in the system (for example the presence of noise). But although the above remarks were made in a different context, their observation that the search space which is traversed during learning in a recurrent network is much more complex than that of a feed-forward network is still useful and important to the examination of the ability of recurrent networks to perform sequence processing tasks. If we consider the landscape metaphor for describing the processes that underlie dynamical systems, attractors are often far more complex than simple fixed points. Given that local minima are a problem in traditional feed-forward networks, the complexity of recurrent network state spaces and the variety of attractors that lie within them would make the journey to the optimum solution much more difficult.

Simplifying the state space by having a sparsely connected architecture does little to improve the performance of the RTRL network. A sparsely connected architecture has fewer degrees of freedom than a fully connected RTRL architecture. This would suggest that the reduction in the dimensionality of the state space does not reduce its complexity sufficiently in terms of the density of local minima, different type of attractors etc. so as to enable learning to take place. This would suggest that the Simple Recurrent Network is better suited to the classification of test data than the RTRL network. It is much more reliable at the task of learning and is able to generalise what it has learned to test data. The Simple Recurrent Network is derived from the standard three layer back propagation network, which it is claimed, is also capable of great power and generality.

The Simple Recurrent Network, although it has full interconnectedness from one layer to the next (with the exception of connections which run from the hidden layer to the context layer), has far fewer connections than an RTRL model. A Simple

Recurrent Network with two input units, three hidden units, three context units and one output unit will have a total of twenty one connections, compared to a RTRL network consisting of two units in the input layer and seven units in the RTRL layer which has sixty one connections.

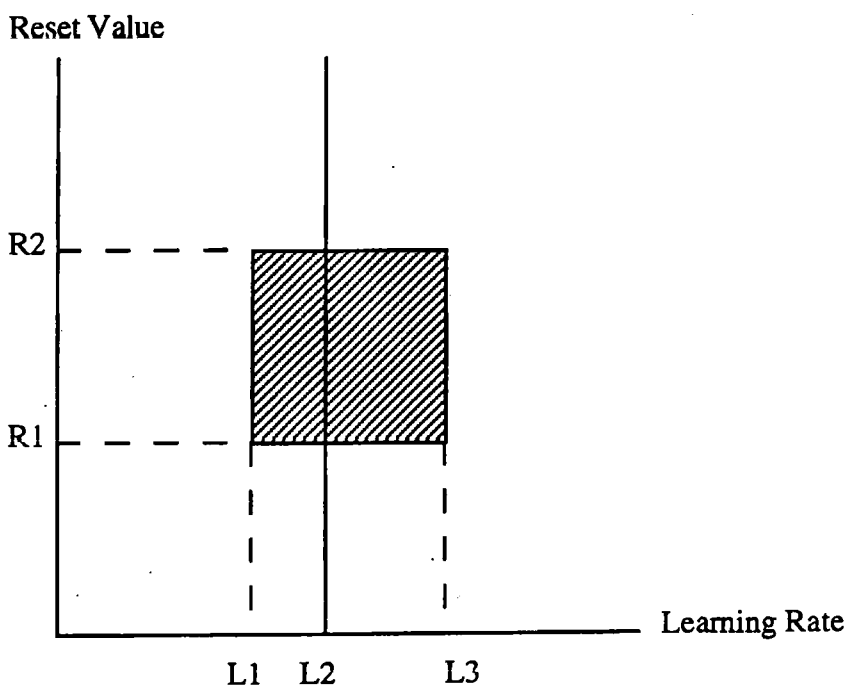
This would suggest that, in order to be effective, recurrent networks should be as sparsely connected as possible. Each additional connection adds one more dimension to the state space which the network traverses whilst attempting to find a solution. As a state space increases in dimensionality, the number of possible network states increases, thus making the "search" for the state (or states) which produce the desired behaviour more complex. This leaves the question of how sparse the connectivity can get before the network is incapable of solving the problem (see section 5.3). Note that a successful use of the RTRL algorithm (Catfolis 1993) did so using networks with relatively few units and a modified form of the algorithm. The only successful use of an RTRL model in the simulations described in chapters three and four used a network with four RTRL units, whereas other tasks used twenty RTRL units. Reducing the number of RTRL units to a similarly low value (five) did not improve network performance, which suggests that overfitting was not the problem, rather the short term memory requirements of the task and the state space it created were too complex for the RTRL algorithm.

The Gamma Model achieves roughly the same rate of success as the Simple Recurrent Network, but with a radically different architecture which allows the network to develop a short term memory mechanism tailored to the task in question. As we saw with the modification of the RTRL algorithm suggested by Catfolis, changing the parameters of the memory mechanism can lead to significant differences in the behaviour of the network and can often mean the difference between success and failure. Thus either the memory mechanism itself finds the optimum parameters for a particular task or some external method is found, such as those suggested in section 5.3.

## 5.2 The Effect of The Learning Algorithm on Learning

On this evidence the RTRL learning algorithm would seem to be a poor choice. It only proved to be able to perform one task (XOR with two step delay) with any sort of consistency, and performed significantly worse than either the Simple Recurrent Network or the Gamma Model on the other three tasks. Some of the reasons for these failings were discussed in chapter four, centring on the idea that the RTRL algorithm does not develop a short term memory mechanism capable of handling sequences of different lengths.

What the RTRL algorithm did demonstrate however is the sensitivity of the learning ability of recurrent networks with regard to changes in certain parameters. If one considers the resetting of  $P_{ijk}$  values discussed in section 4.3 and the effect that this has on the choice of learning rate it is clear that, for a given sequence processing task, there will be a range of  $P_{ijk}$  reset values which allow the use of higher learning rates (see figure 5.2)



*Fig 5.2: Graph showing the relationship between learning rate and  $P_{ijk}$  reset values*

In Figure 5.2 testing the two different parameters over a hypothetical sequence processing task has shown that if the  $P_{ijk}$  values are never reset, the network is able to learn the sequence for values of the learning coefficient  $K$  such that  $K$  is less than  $L_2$ . However, if  $P_{ijk}$  is reset after a number of trials such that the reset value  $r$  has a value between  $R_1$  and  $R_2$  the network is able to learn, furthermore the learning may well be faster between the coefficient values  $L_2$  and  $L_3$ , where learning would have been less reliable when periodic  $P_{ijk}$  resetting had not been in place. Unfortunately this did not increase the learning ability of the RTRL network so that it would be on a par with the other two models tested.

One way to improve learning in neural networks generally is by the use of noise. Amit (1992) makes the point that if the level of noise in a system is too high, there is more chance of the network "stepping in the wrong direction". However, some levels of noise may aid learning:

"At higher levels of noise [the network] may hop across barriers between adjacent minima. In this way noise may be an agent for eliminating the effect of spurious states while preserving the retrieval in the stored memories...the barrier for crossing from a local minimum to a global one is lower than the barrier for the reverse process. This promises that there be a window in noise values for which spurious states be de stabilised, while the stored memories remain good attractors." (Amit 1992 pp 86-87)

In other words, noise allows the network to leave local minima (what Amit refers to as "spurious states"). However, it is also possible that too much noise could cause the network to jump out of either the global minimum or a local minimum which represents a tolerably low error score.

The method of resetting  $P_{ijk}$  values as examined in section 4.3 would appear to fit into this pattern. Similarly, Elman (1993) uses periodic zeroing of the context

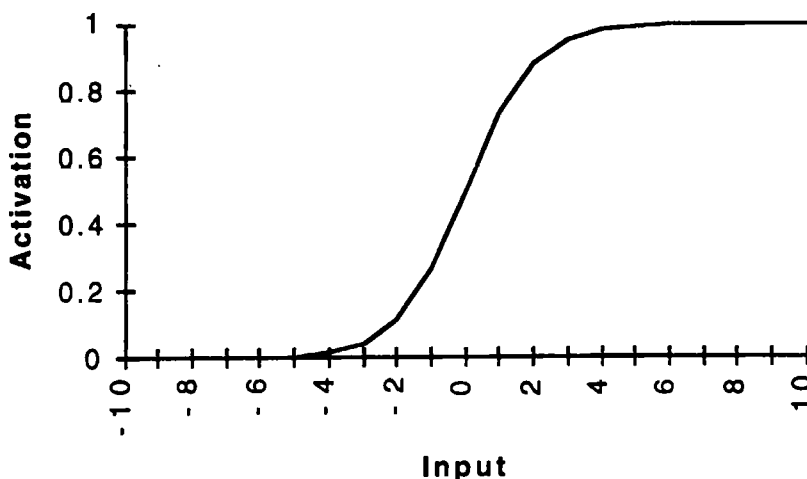


unit outputs to improve learning. In a word in sentence prediction task (the network was given a sentence one word at a time and had to predict the next word in the sentence) learning was divided into a number of phases:

- i) Zero the output of the context layer randomly after either every third or every fourth word.
- ii) Increase the interval between resets to four or five words (again the exact value was chosen at random).
- iii) Increase the interval between resets to five or six words
- iv) Increase the interval between resets to six or seven words
- v) Do not reset at all.

Elman found that by using this method the network was able to learn sequences that it was unable to if learning had taken place without zeroing of context unit outputs. It was claimed that this approach was analogous to learning with graded data sets (see section 5.4). Note however that the period between setting of context layer outputs to zero is not fixed and is increased during learning before finally doing away with the process altogether.

The reason for this flexibility is due to the nature of the activation function used in neural networks. In figure 5.3 we can see that the range of greatest sensitivity is when the input to a unit is around zero.



*Figure 5.3: A typical logistic function used to determine neuronal activation. Note how the same difference between two sets of inputs can lead to much greater differences in output. The difference in output between an input of 0 and 5 is much greater than between inputs of 5 and 10. (The sigmoid function,  $1/(1+e^{-x})$ , is shown).*

At the start of a typical learning regime, connections are randomly determined between a small range. Thus the net input to a unit will tend to be close to zero, with the result that at the start of its journey through the state space the network will be in the region of greatest sensitivity. As the network learns and connection strengths become more strongly excitatory or inhibitory the net input may well move away from this sensitive region and changes in the level of activation become harder to achieve. A unit will start to receive inputs which are much more strongly excitatory or inhibitory, with the result that unit outputs will be pushed towards less sensitive regions of the curve. Thus it is much more difficult to cause radical changes in the outputs of a network during the later stages of the learning process.

The periodic resetting of unit activations in the context layer to zero (or resetting  $P_{ijk}$  values in the case of the RTRL model) will also serve to move the network towards this more sensitive region of the state space, since resetting causes a reduction in net input to units in the hidden layer. This means that there is an

increased level of flexibility in the system and that the loss of plasticity that occurs during the later stages of learning is reduced. Note that the nodes in an RTRL network will not be moved towards this sensitive region since the  $P_{ijk}$  variable does not play a part in determining the activation or output of a node.

### **5.3 What does this mean for efficient Learning?**

The discussion in sections 5.1 and 5.2 above might well be summarised as follows: whether or not we succeed, or get trapped in a local minimum depends on a number of factors, including how big the steps are through the weight space which we allow ourselves, as well as what the shape of the error surface looks like "because the error surface is a joint function of the network architecture and the problem at hand" (Elman 1993 pp91).

In the above quote, Elman takes the word "architecture" to mean both the physical layout of the network (number of layers, number of units per layer etc.) as well as the learning algorithm. The physical layout of the network determines the dimensionality of the state space, whilst the learning algorithm determines the way in which the network moves through the state space.

Thus changing the parameters of the learning rule and modification of the network architecture can significantly effect the ability of a neural network to perform sequence processing tasks. The picture is further complicated by the fact that one set of network parameters may work well on one task, but not on others. If we wish to use a neural network to perform such a task one would ideally like a way to search through the various network configurations without having to laboriously wade through them by hand i.e. testing lots of different types of network parameters until you find the best one. Techniques to automate this process have been developed, one of which will be described below.

### 5.3.1 Adaptive Memory Mechanisms

The main way in which recurrent networks differ from feed forward networks is their ability to develop a short term memory mechanism which allows them to perform sequence processing tasks. Different tasks require different types of memory: some sequences will be of fixed length, others will be of variable length. The output at a certain time may depend on short or long term factors, or a combination of the two. It is because of this variability that an adaptive memory mechanism is desirable, since it can modify its internal parameters to suit the particular task in question.

The limitations of static memory mechanisms were set out by Mozer (1993): "Static memory models can be a reasonable approach if there is adequate domain knowledge to constrain the type of information that should be preserved in the memory". However many problem domains may not have sufficient knowledge known about them for this to take place. This means that adaptive memory mechanisms are more desirable since they can overcome the lack of knowledge by building a tailored short term memory mechanism during learning.

This class of short term memory mechanisms is under exploited by researchers. At present the most popular type of adaptive memory mechanism is de Vries and Principe's gamma model, which was described (along with modifications suggested by other researchers) in chapter two (see section 2.1.8 for further details). An area which is relatively unexplored however is the use of Gamma Memory structures with different types of content. Mozer (1993) describes six different types of memory content (see chapter two section 2.3 for further details). Gamma memory networks to date have contained the activations from within the memory structure from the previous time step. However it is possible to use a non linear transformation function on either the current input to the network or the contents of the short term memory mechanism.

Furthermore the memory could hold a completely different type of content such as the output of the network from the previous time step. Mozer states that the latter type of memory content would be particularly useful for auto predictive tasks i.e. one where the network is given an input at time  $t$  and the desired output is the input at time  $t+1$ . Given that all but one of the tasks used in this comparative study are of this type<sup>1</sup> (and this type of prediction task is quite a common one in general) it would seem that this type of memory mechanism would be particularly useful.

#### **5.4 Can we improve Learning without changing the network?**

In the above sections we have looked at various means of improving the ability of recurrent neural networks to perform sequence processing tasks. This has been attempted by modifying some aspect of a particular network architecture or learning algorithm. However, it is worth noting that there is another possibility for improving network performance: by paying closer attention to the data sets on which the network is trained and tested.

One such method was proposed by Elman (1993). In the same study in which he examined the effect of periodically zeroing the outputs of the context layer he also looked at the effect of grading data sets on learning performance. The Simple Recurrent Network model used proved to be a poor performer on a data set which consisted of simple and complex sentences all mixed together. A new training method was developed. This consisted of five separate databases containing different

---

<sup>1</sup>The obvious exception is the two step delay XOR problem, where the desired output of the network at time  $t$  is the solution to the input pair shown to the network at time  $t-2$ .

proportions of simple and complex sentences<sup>2</sup>. Each database was presented to the network five times and was then discarded. The contents of each database were as follows:

- i) 10,000 simple sentences.
- ii) 7,500 simple sentences and 2,500 complex sentences.
- iii) Simple and complex examples in equal measure.
- iv) 2,500 simple sentences and 7,500 complex sentences.
- v) 10,000 complex sentences.

Whereas in the first training regime the network failed to learn the task, presenting graded data sets allowed the network to learn. This finding was independent of learning rate, initial conditions, number of hidden units etc.

Elman attributes the success of what he refers to as incremental learning to the fact that grading data sets allows the network to organise its state space. As we have seen, learning in neural networks involves the journey through a state space of  $n$  dimensions (where  $n$  is the number of connections in the network) from a random starting point to a point where the network can perform a task perfectly or within some margin of error. When the network starts learning simple sentences the portion of the state space which yields a satisfactory solution is much smaller than in non incremental learning, since only three of the four sources of variance are present<sup>3</sup>. This method of state space restriction also applies when the short term memory

---

<sup>2</sup>In this problem, complexity stems from the fact that sentences sometimes contained "multiple embeddings in the form of relative clauses (in which the head could either be the subject or object of the relative clause). Complex sentences contained relative clauses, simple ones did not" (Elman 1993).

<sup>3</sup>Grammatical category, number of words in sentence and verb argument type are present. Long distance dependencies are only found in complex sentences.

mechanism is restricted (see section 5.2). Either way the network learns a set of subgoals which can then help to guide it towards the final goal later in learning, avoiding local minima at the same time. In learning with non graded data sets, the solution space is very large and may contain lots of different types of attractor, many of which will be local minima.

Elman's findings also lead to questions about the problem of selecting data sets in general. Whilst a small sample size increases the risk that the sample will not be a good representation of the population as a whole, larger data sets may increase the risk that the network will have a restricted ability to generalise (since ambiguity decreases as sample size increases). Elman also proposes that neural networks are at their most flexible during the early stages of learning because of this ambiguity.

## **5.5 Summary**

In this chapter an attempt has been made to draw together the results of the experiments described in chapters three and four, together with the aims of the thesis set out in section 1.4. The conclusions drawn are as follows:

- Architectures which are more sparsely connected are better than those which are more fully connected. The fully connected RTRL network is the poorest performer in the comparative study described in chapter three, learning and generalisation in this model was significantly worse than either the Simple Recurrent Network or the Gamma Model.
- At the same time the network needs to be of sufficient complexity to be able to capture the properties of the data it is trying to learn. Because of this a fully connected RTRL network is sometimes able to outperform a sparsely connected RTRL network. For example see the comparison between a fully connected and

sparse RTRL network over the finite state grammar task (Chapter Four Table 4.6).

- Once the network is able to learn a data set, increasing its complexity does not improve learning further. If anything increasing architectural complexity will lead to a poorer performance. For example see the comparison between a fully connected and sparse RTRL network over the XOR with two step delay task (Chapter Four Table 4.4).
- Learning rules which are non-local perform worse than those learning rules which change according to local values only. For example the RTRL algorithm which is non-local is outperformed by the Simple Recurrent Network and the Gamma Model, whose learning rules are more local (see chapter three).
- Restricting the memory capacity of a learning rule (by periodic resetting of the  $P_{ijk}$  values of the RTRL algorithm for example) can lead to an improvement in performance over the same architecture. For example the superior performance of the Catfolis implementation of the RTRL algorithm against the original RTRL algorithm described in chapter four section 4.3.
- Evidence from sections 4.2 and 4.3 would seem to suggest that the learning rule has the greater effect on network performance. The improvements in the performance of the RTRL algorithm were greater when the learning rule was modified than when the architecture was modified. The modified learning rule performed significantly better over two data sets (see Chapter Four tables 4.12 and 4.13) as against one for the RTRL network with modified architecture (Chapter Four Table 4.4). Moreover, at no time did the original RTRL algorithm outperform the modified learning algorithm. This was not the case with the modified architecture (see third point above).
- The fact that modification of the RTRL networks architecture or learning rule alone does not raise the level to that of the Simple Recurrent Network or the Gamma Model suggests that recurrent networks should be relatively sparsely connected and have a non local learning rule to be most effective.



## **Chapter Six. Further Work**

### **6.1 An Aside on Genetic Algorithms**

Neural networks are not the only type of computer program which adapt themselves to perform a particular task. There exists another class of programs called Genetic Algorithms which do this as well. Just as neural networks can be said to "learn" to perform a particular task, genetic algorithms can be said to "evolve" to perform a particular task. Genetic algorithms can be defined as follows:

"Genetic algorithms are search algorithms based on the mechanics of natural selection and natural genetics. They combine survival of the fittest among string structures with a structured yet randomised information exchange to form a search algorithm with some of the innovative flair of human search." (Goldberg 1989 pp1)

What does this mean in practice? Assuming that we want to perform a particular task, the first step is to create a set of "creatures" (in our case computer programs) to perform the task. The characteristics of each creature can be encoded in a binary string, analogous to human DNA. When using genetic algorithms, each string can be viewed as a single approach to tackling a particular problem. Each bit within the string represents various objects which are important to the approach represented across the string as a whole. Let us take as a simplified example<sup>1</sup> a population of four strings:

1, 01101

2, 11000

3, 01000

---

<sup>1</sup>This example is taken from Goldberg (1989)

4, 10011

Each string then attempts to perform the task set. Its success (or otherwise) in doing this is quantified by a fitness function, as shown in table 6.1

String Number	String	Fitness	% of Total
1	01101	169	14.4
2	11000	576	49.2
3	01000	64	5.5
4	10011	361	30.9
Total		1170	100.0

*Table 6.1 Sample strings and fitness values*

Assuming that none of the strings above represent a satisfactory solution, a new generation of creatures are needed. These are created by using the following operators:

**Reproduction:** Strings become involved in the reproduction process by means of their fitness score. Sufficient numbers of strings are chosen until there are enough of them to breed a new generation. The probability of a string being chosen is equal to the percentage of total fitness it achieved. In our example, string 2 would have a probability of .492 of becoming involved in this process.

**Crossover:** This is the means by which new strings are created. A point along the string length is randomly chosen. If we have a string population of length  $l$ , partitioned at point  $p$ , the first new string will contain its original code up to point  $p$  and the portion of the second string between  $p+1$  and  $l$ . If in the above example

strings 2 and 4 were chosen for crossover, with the partition between the third and fourth bit the process would look like this

<i>Old Strings</i>	<i>New Strings</i>
2 = 110   00	2' = 11011
4 = 100   11	4' = 10000

Mutation: This is a simple operation involving random "flipping" of bits. If mutation were to be applied to string 4' for example (and the third bit was randomly selected for mutation) the result would be as follows:

<i>Before Mutation</i>	<i>After Mutation</i>
4' = 10000	4' = 10100

There is much debate in the genetic algorithm literature concerning the usefulness of different evolutionary processes, the contribution of mutation to the evolutionary process and so on. The point here is to illustrate the general principles of genetic algorithms and their operation.

### **6.1.1 Genetic Algorithms and Recurrent Networks**

There has been much crossover between research into genetic algorithms and neural networks, using the former to improve the performance of the latter. This has been attempted in two ways: firstly by using a genetic algorithm as learning rule. Secondly by using a genetic algorithm as a means of modifying the network architecture. An example of these methods is the GNARL<sup>2</sup> algorithm created by Angeline, Sanders and Pollack (1994).

---

<sup>2</sup>The acronym stands for GeNeralised Acquisition of Recurrent Links.

The algorithm works as follows: a population of networks are randomly generated and tested. The fittest 50% are selected to breed the next generation (see section 5.3.1). The fitness function is some error score when the actual output of the network is compared against the desired output. Angeline, Sanders and Pollack quote three different error scores: sum of square errors, sum of absolute errors and sum of exponential absolute errors. However they claim that the choice of fitness function does not effect the mechanics of the algorithm. During the breeding process, the level of mutation is determined by how close the parents are to being a solution to the task. Networks which are far away from a solution are more likely to undergo severe mutation. Conversely, networks which are close to a solution are more likely to undergo slight mutation. Thus the search undertaken by the algorithm is coarse grained to begin with, but becomes more fine grained as it gets closer to a solution. This is defined by a variable called the temperature of the parent  $T(\eta)$ :

$$T(\eta) = 1 - \frac{f(\eta)}{f_{\max}} \quad (6.1)$$

Where  $f(\eta)$  is the fitness level of the parent and  $f_{\max}$  is the maximum fitness for the task.

Weight updates are accomplished on a method based on perturbation of connection strengths with gaussian noise. However GNARL's update algorithm compensates for the tendency of this method to inhibit the offspring's ability to outperform its parent. Firstly the instantaneous temperature of the network is computed:

$$\hat{T}(\eta) = U(0,1)T(\eta) \quad (6.2)$$

Where  $U(0,1)$  is a uniform random variable between 0 and 1. This term is then substituted into the following weight update rule:

$$w = w + N(0, \alpha T(\eta)) \quad \forall w \in \eta \quad (6.3)$$

Where  $\alpha$  is constant and  $N()$  is a gaussian random variable.

Structural changes are concerned with changing the number of hidden units and the level of connectivity between all nodes. In order to avoid radical changes to the network new connections have an initial value of zero and new units have no incoming or outgoing connections. These features are added in future generations.

$$\Delta_{\min} + \left[ U[0,1] \hat{T}(\eta) (\Delta_{\max} - \Delta_{\min}) \right] \quad (6.4)$$

Where  $\Delta_{\max}$  is the maximum number of nodes or links added or deleted and  $\Delta_{\min}$  is the minimum number of nodes or links added or deleted.

This system was trained on a set of regular languages (See table 6.2). Each language was shown to the network as a series of positive and negative examples.

Language	Description
1	$1^*$
2	$(1,0)$
3	No odd length 0 strings anytime after an odd length 1 string
4	No more than two 0's in a row
5	An even number of 10's and 01's pairwise
6	$(\text{Number of 1's} - \text{number of 0's}) \bmod 3 = 0$
7	$0^*1^*0^*1^*$

*Table 6.2. Languages learned by the GNARL algorithm. \* indicates that a character can be either a 0 or a 1.*

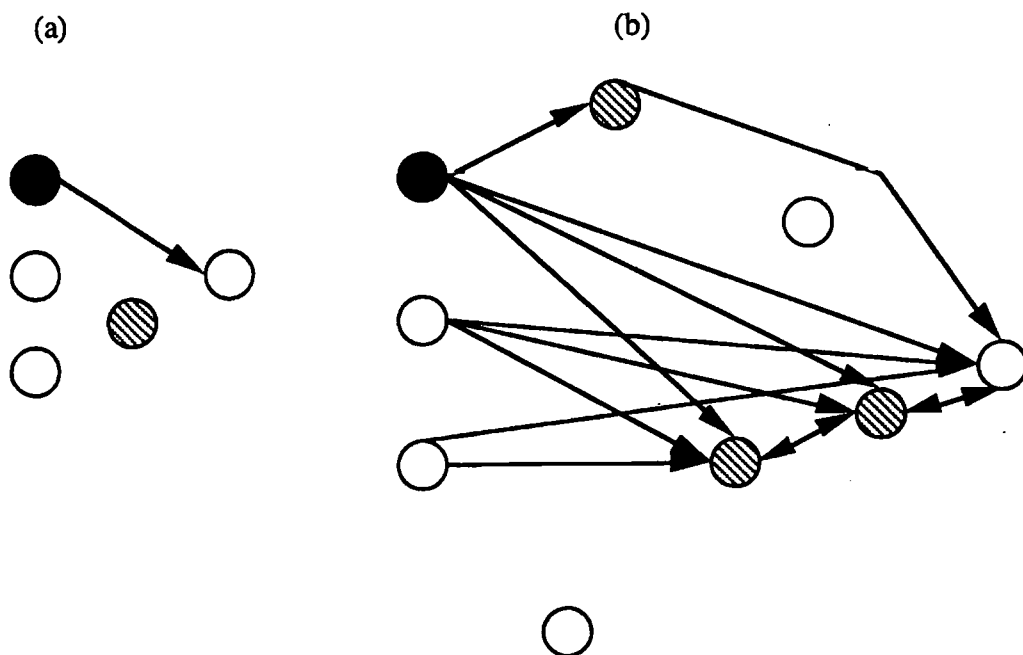
Two sets of experiment were performed. One using sum of absolute errors as a fitness measure, the other using sum of square errors as a fitness measure. The results are shown in table 6.3.

Language	Evaluations (SAE)	% Accuracy (SAE)	Evaluations (SSE)	% Accuracy (SSE)
1	3975	100.00	5300	99.27
2	5400	96.34	13975	73.33
3	25050	58.87	18650	68.00
4	15775	92.57	21850	57.15
5	25050	49.39	22325	51.25
6	21475	55.59	25050	44.11
7	12200	71.37	25050	31.46

*Table 6.3: Speed (Number of generalisations) and accuracy of the GNARL method over the languages described in table 6.2*

Unsurprisingly, the algorithm was most efficient at learning the simplest grammar from the set (number one). This was the case both in terms of the number of generations needed to evolve to an optimum solution and the percentage accuracy of said network on the task.

An additional feature of networks evolved using the GNARL algorithm is that the complexity of the networks increased as they evolved. An example of this can be found when GNARL was used to perform what was termed an "Enable-Trigger Task". This task involves the following rule. For two inputs (a,b) and a starting state S1, the network switches to state S2 when a=1 and remains in this state until it is triggered by b=1, when the network has a desired output of 1 and reverts to state S1. For example the input stream [(0,0) (0,1) (1,1) (0,1)] will have the desired output [0, 0, 0, 1]. The fittest members of two generations are shown in figure 6.1.



*Figure 6.1: Architectures of two networks evolved by the GNARL algorithm to solve the "enable-trigger task". Black circles represent bias units, shaded circles represent units which have feedback connections to themselves.*

In the above diagram (a) represents the fittest architecture of generation 1. (b) represents the fittest architecture of generation 765. This network solves the task for all strings of length eight. Note that even at generation 765 the network is still evolving since we can see that two units have no connectivity. This is because one of the rules for breeding new offspring states that units which are added at generation  $g$  may not be connected until generation  $g+1$  at the earliest.

## 6.2 Some Guiding Principles for Recurrent Network Development

In terms of the recurrent networks themselves, learning rules where the change in one weight is affected only by local considerations gives a learning and generalisation performance at least as good as non local learning rules such as RTRL,

and has the added advantage of being less computationally expensive. Following on from that this thesis has shown what has been stated elsewhere in the literature, that recurrent networks whose architectures are complex (lots of units, full interconnectivity etc.) are less efficient than their more sparsely connected relatives.

The key to the complexity of a recurrent network appears to lie in its hidden layer where, during learning, the network attempts to develop an internal representation of some particular task. If the hidden layer is too small then the network will not be able to develop this internal representation. If the hidden layer is too large then although the network will theoretically be able to develop this internal representation, the state space generated will be very complex, with a much higher probability that the network will get stuck in a local minimum during learning.

It would appear that if there is a single thread running through this thesis it is the relationship in successful learning systems between simplicity and complexity. Given an identical size in terms of the number of hidden units, the networks examined in chapters three and four differ considerably in terms of the complexity of their state spaces. Let us take two hypothetical recurrent networks N1 and N2, where the state space of N1 is less complex than the state space of N2. If N1 and N2 both have state spaces that are rich enough to learn a given set of input - output relations, then N1 is more likely to be successful at actually learning this set of relations. This is because the more complex state space of N2 means that the network is more likely to encounter local minima during learning.

Thus the failure of the RTRL network to perform as well as the Simple Recurrent Network or the Gamma Model is not because its state space is insufficiently complex. Indeed so great is the problem of having an overly complex state space that improved methods of state space traversal (i.e. using more powerful learning algorithms) are not sufficient to overcome this handicap.

Similarly Elman's (1993) set of experiments concerning the Simple Recurrent Network demonstrated how learning of simple concepts enabled the network to learn more complex concepts which might have otherwise been beyond its comprehension



had the order of learning stayed random as in most neural network learning regimes. Similarly Angeline, Saunders and Pollack (1994) showed that the recurrent networks evolved by their genetic algorithm system started as simple networks, but became more complex as their evolution progressed. When during the evolutionary process complex and more simple architectures fought it out the simpler architecture always came out on top.

Comparative studies described here and elsewhere show that recurrent networks which use adaptive memory mechanisms (e.g. deVries and Principe's Gamma Model) are perhaps the most promising avenue of research, since they give rise to networks which are biologically plausible (Bressloff 1993) and satisfy the problem of attempting to develop an appropriate short term memory mechanism for a particular task (since the network does this for us). When using the gamma memory structure, an interesting question might be just what should the outputs of the memory structure feed forward to? The search for biological plausibility and the results of the comparative studies carried out would suggest that such a structure would be sparsely connected and perhaps modular in design, such as the CALM algorithm developed by Murre (1992). Associated with such architectures are learning rules where weight changes in one particular part of the network are the function of local information only.

Although these guiding principles will be useful rules of thumb for researchers to create new recurrent network models or improve on old ones, the sheer diversity of recurrent network types means that the search space to be explored is still very large. However new techniques, particularly genetic algorithms, can be used to automate the process. Accordingly, when neural networks are being developed, it is no longer sufficient to talk of training as being the change in connection strengths by a predetermined learning rule between units in an otherwise fixed architecture as being the whole picture.

The two approaches outlined above represent processes where the system starts simple and goes complex. Of course it is possible that starting from a position

of complexity and moving towards simplicity will give rise to a satisfactory solution. This route is represented by learning algorithms which represent pruning strategies. However, the first approach (from simplicity to complexity) is superior since with a pruning algorithm learning may not succeed because the state space of the existing network is too complex or because the state space is not complex enough (i.e. there are too few hidden units). Thus the guiding principle for recurrent network development would appear to be this: Start simple, then get as complex as you need to be, but no more.

### **6.3 Speculations (i): A New metaphor for Recurrent Network Training**

All neural networks attempt to solve a particular problem by learning i.e. by forming an internal representation of the data with which they have been presented. Neural networks have proved to be a powerful technique for solving many different types of non-trivial problem. However this study has shown that the parameters of the network can often determine if learning has been successful or not. A given application may fail, not because a neural network is unable to perform a particular task, but because the wrong parameters were chosen. Furthermore the nature of the data set used can also increase or decrease the probability of success or failure.

It therefore seems appropriate to take on board these results and integrate them into a new metaphor for developing neural networks, whereas previously the metaphor has been one based on learning, the proposed metaphor is based on a notion of agency. The definitions of agency are many and varied. Here an agent is deemed to be an object, operating in an environment such that it can understand aspects of its environment and can generalise these to novel situations. Agent is a broad term which can include both natural (e.g. humans, animals) and artificial objects (e.g. robots).

Instead of speaking of a neural network learning a given task the agent metaphor instead speaks of development. This can be broken into three connected stages:

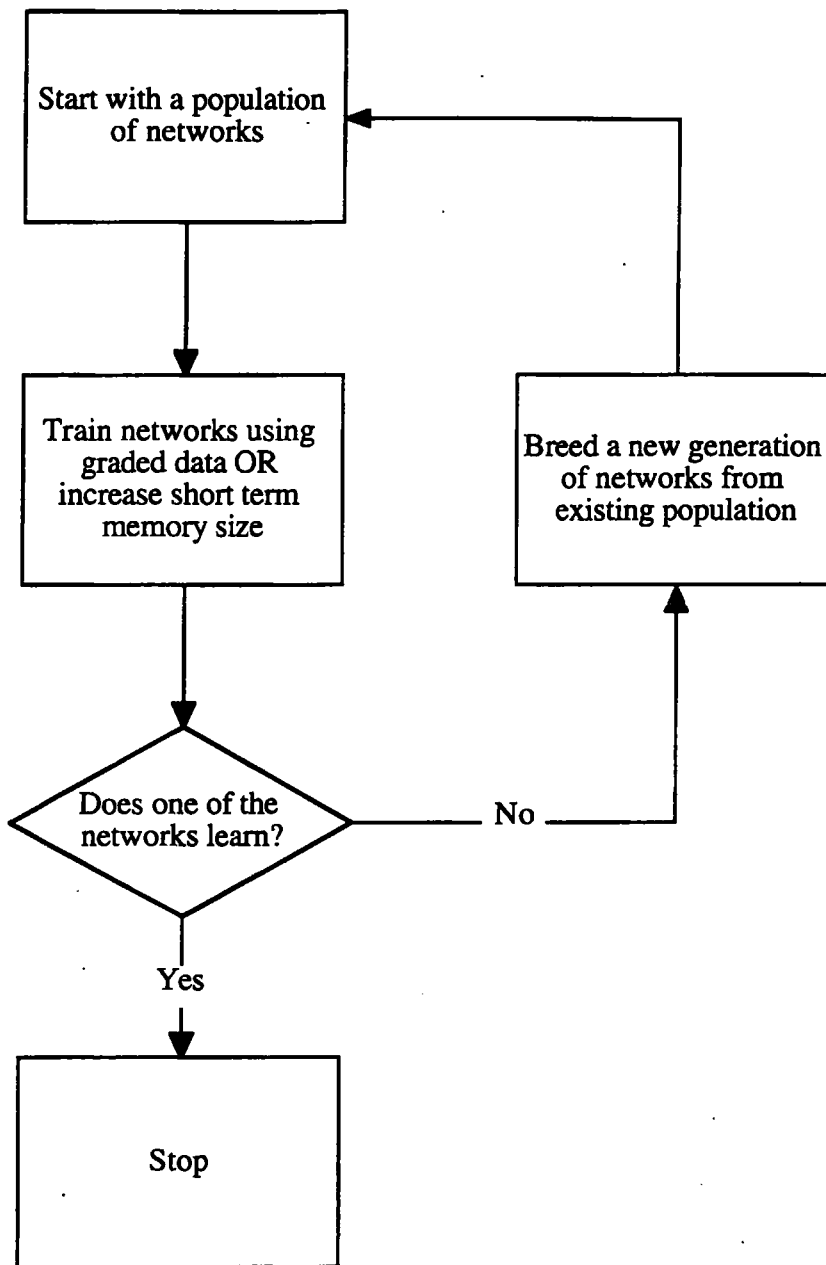
1. Evolution: In the real world no agent starts from scratch. Almost all animals have non-learned behaviours (instincts) which are encoded in its genes and are the result of the evolutionary process. The same could also be said for the form of its body, which is adapted so that an agent is best able to move around the environment, spot dangers and exploit whatever resources are available. In the same way genetic algorithms could be used to determine the architecture (and/or the connection strengths) of a neural network. Research has shown that genetic algorithms can be used to get round some of the problems found when using recurrent networks (Angeline, Saunders and Pollack 1994) and are an elegant way of finding the optimum parameters of a network.

2. Tuition: In a world of intelligent agents, it is rare for a new agent to be left to find its way in the world without help from more experienced peers. Humans learn complex concepts with the help of others, starting with relatively simple concepts, then progress to more complex ones, using the learned simple concepts as building blocks. The work of Elman (1993) shows that this approach can also be extended to recurrent networks, where a network that was presented with graded training data was able to learn a task in which it was unable to learn the same, ungraded, data.

3. Training: This is the traditional learning rule based part of the process. This may or may not include restricting the short term memory mechanism. As Elman (1993) noted, presenting graded training data has the same effect as restricting the short term memory mechanism. The precise choice will largely depend on the "gradability" of the data set. If the data is easily gradable ( for example in Elman's study, complexity was judged by the presence or otherwise of embedded clauses in a sequence of

words) then this is an option. However the alternative method of restricting the short term memory mechanism is more attractive for two reasons: not all data is easily classified in this way, and restricting the short term memory mechanism offers a way of using this technique on data whose complexity is both easily or not so easily defined. Furthermore restricting the short term memory mechanism can be seen to be analogous to the modifications to real brains in the formative stages of their development, and is therefore more plausible from a physiological perspective.

This new metaphor gives rise to the development cycle displayed in figure 6.2:



*Figure 6.2: A new development cycle for recurrent neural networks based on Agent Theory.*

Note that although the agent theory metaphor may apply there are in fact a number of different development cycles that could be tried (see section 6.5).

According to the agent metaphor the recurrent network is a creature attempting to perform a particular task. The ability of the network to perform the task is a measure

of its evolutionary fitness. Furthermore the short term memory mechanism of the network grows more powerful during learning.

### **6.3.1 The New Metaphor and Amit's Criteria**

In chapter one, a shopping list of the desirable properties of a recurrent network was set out by Amit (1992). How does the agent theory metaphor stand up to some of these demands?

*Biological Plausibility:* The agent metaphor is a much more holistic approach than traditional neural network learning paradigms. Taking into account evolution (the use of genetic algorithms) and allowing expert knowledge to facilitate development (grading of data sets), Happel and Murre (1994) showed that neural networks evolved using genetic algorithms often give rise to architectures which resemble some aspect of real brains, in their case "The best performing network architectures seem to have reproduced some of the overall characteristics of the visual nervous system". Also the use of genetic algorithms seems to lead to better performance (see the "Emergent Behaviour" section below) and is cited by Happel and Murre as to why "for many vital learning tasks in organisms only a minimal exposure to relevant stimuli is necessary" (Happel and Murre 1994 p 985).

*Emergent Behaviour and Potential for Abstraction:* These two categories have been linked together because of the high degree of overlap. A key aspect of the emergent behaviour of a neural network is the ability of the network to successfully perform a task on novel data (i.e. its potential for abstraction). The guidelines suggested for developing networks in the future (sparsely connected networks, local learning rules, adaptive short term memory mechanisms) give rise to better generalisation abilities. Furthermore, using genetic algorithms to develop network architectures "can not only enhance learning and recognition performance, but can also induce a system to better

generalise its learned behaviour to instances never encountered before" (Happel and Murre 1994 p 985).

*Freedom From Homunculi:* This criterion is concerned with the need for the system to adapt itself to a particular task without the need for some kind of overseer. It could be argued that using genetic algorithms, grading data sets and restricting the short term memory mechanism during the early stages of learning are forms of homunculi. However, in developing a neural network in these fashions, the parameters of the network are set implicitly, not explicitly. It may well be that an architecture evolved through the use of a genetic algorithm will be sparsely connected, but this is because the dynamical properties of the network mean that such an architecture is best suited to the particular task, not because it has been explicitly programmed to do so. Similarly it is because of the network dynamics that restricting short term memory and grading the data set gives rise to improved performance. There may be problems where such an approach is less necessary.

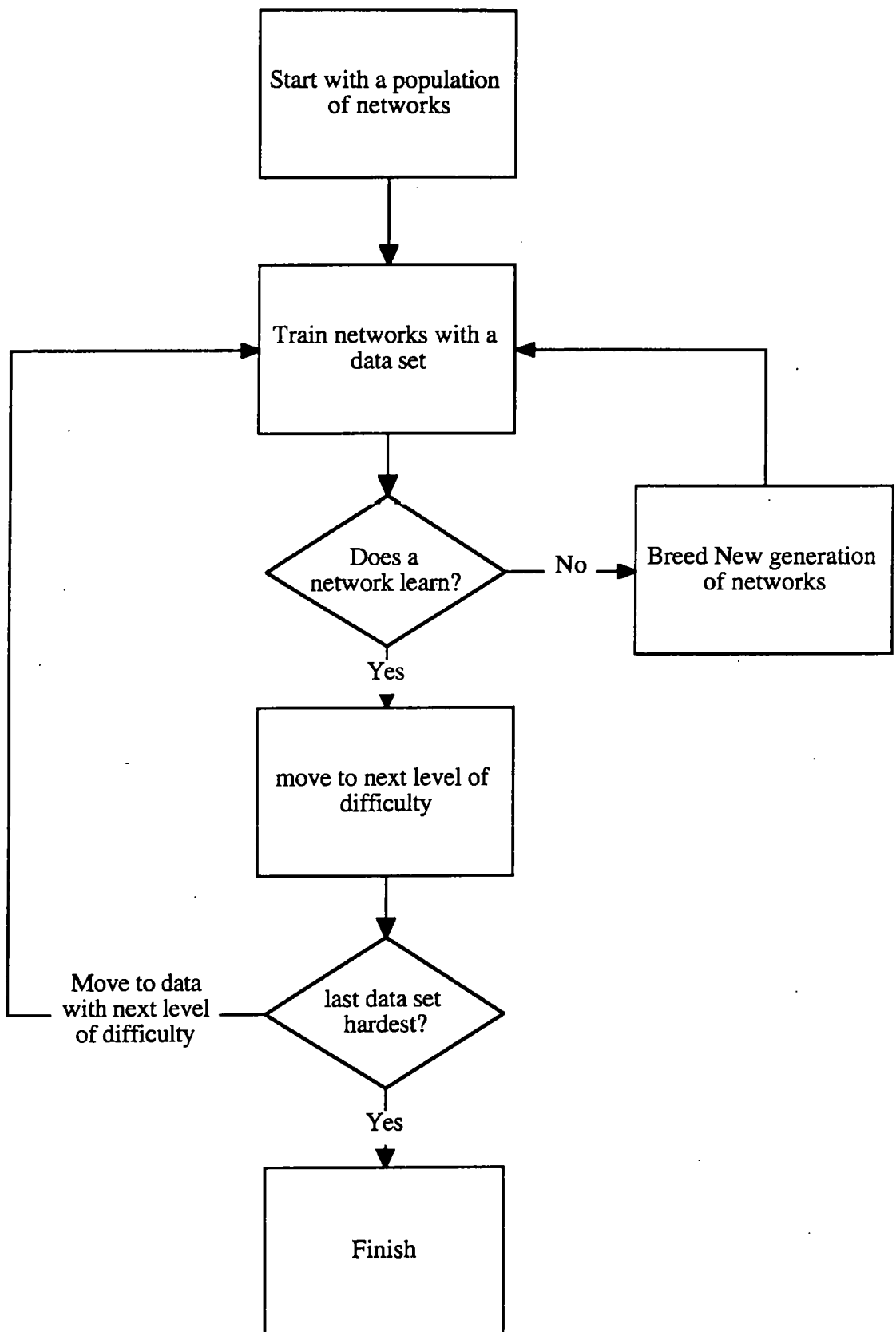
*Parallel Processing Hardware:* A neural network will need to have a high degree of parallel processing if it is not to take up an unreasonably large amount of computing time to perform a particular task. Furthermore the ability of real brains to perform non trivial tasks very quickly means that this is an important requirement if we are particularly interested in making the network as physiologically plausible as possible.

*Associativity :* This refers to the ability of a network to combine similar inputs into a single representation for the purposes of cognition. Taking an example from visual processing, an individual object may look different when viewed from different angles, but it is the same individual and needs to be recognised as such. As with parallel processing, a neural network will need to have this ability if it is to be used to perform non trivial tasks. Again it is an important ingredient of physiological and psychological plausibility.

## **6.4 Speculations (ii): Future Research**

The most obvious avenue of future research is to use the theoretical framework in section 6.2 in a system for creating recurrent networks. Attempting to do so immediately creates questions: for example the relationship between the evolution of network architecture and the presentation of graded data sets. One way to do this is shown in figure 6.2. However other combinations are possible, as shown in figure 6.3. Here network fitness is judged at each level of difficulty, so that a random sample of networks are the starting population at the first level of difficulty, the fittest networks at the first level of difficulty are the starting population at the second level of difficulty and so on. This cycle continues until all data sets have been successfully learned. Thus the philosophy of starting with simple problems is also applied to the evolution of the network architecture as well as connection strengths.





*Figure 6.3 An alternative development cycle for recurrent neural networks based on Agent Theory.*

An additional area of research would be to examine the extent to which the state of a recurrent network is determined by the genetic algorithm or by the learning rule. Is the genetic algorithm simply to be used to create the architecture with no role in determining connection strengths, does the genetic algorithm determine connection strengths alone or is a more hybrid approach required? Another question is concerned with the overall plasticity of the system: is the whole network modifiable by genetic algorithm or is it only part modifiable (for example we might wish to use a particular type of memory structure, but the rest can be created by the evolution of the network)?

What these questions do demonstrate is that recurrent neural networks represent a powerful tool for sequence processing tasks and that their limits have not yet been discovered. It is clear that much research still needs to be done.

## References

- Alexandre, F., Guyot, F., & Haton, J.-P. (1990). The Cortical Column: A New Processing Unit for Multilayered Networks. *Neural Networks*, 3, 15-25.
- Amit, D. J. (1992). *Modeling Brain Function: The World of Attractor Neural Networks*. Cambridge University Press.
- Angeline, P. J., Saunders, G. M., & Pollack, J. B. (1994). An Evolutionary Algorithm that Constructs Recurrent Neural Networks. *IEEE Transactions on Neural Networks*, 5, 54-65.
- Back, A. D. & Tsoi, A. C. (1991). FIR and IIR Synapses, a New Neural Network Architecture for Time Series Processing. *Neural Computation*, 3(3), 375-385.
- Bressloff, P. C. (1993). Temporal Processing in Neural Networks With Adaptive Short Term Memory: A Compartmental model Approach. *Network Computation In Neural Systems*, 4, 155-175.
- Catfolis, T. (1993). A Method for Improving the Real-Time Recurrent Learning Algorithm. *Neural Networks*, 6, 807-821.
- Cleeremans, A. & McClelland, J. L. (1991). Learning the Structure of Event Sequences. *Journal of Experimental Psychology: General*, 120, 235-253.
- Cleeremans, A., Servan-Schreiber, D., & McClelland, J. L. (1989). Finite State Automata and Simple recurrent Networks. *Neural Computation*, 1, 372-381.

Crick, F. H. C. & Asanuma, C. (1986). Certain Aspects of the Anatomy and Physiology in Cerebral Cortex. In J. L. McClelland & D. E. Rumelhart (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition Volume 2: Psychological and Biological Models* (pp. 333-371). M.I.T Press.

Dayhoff, J. E., Palmadesso, P. J., & Richards, F. (1994). Developing Multiple Attractors in a Recurrent Neural Network. In *Proceedings of World Congress on Neural Networks*, 4 (pp. 710-715). Town & Country Hotel, San Diego, California, USA.: Lawrence Earlbaum Associates.

De Vries, B. & Principe, J. C. (1992). The Gamma Model - A New Neural Model for Temporal processing. *Neural Networks*, 5, 565-576.

Elman, J. L. (1990). Finding Structure in Time. *Cognitive Science*, 14, 179-211.

Elman, J. L. (1991). Distributed Representations, Simple recurrent Networks, and Grammatical Structure. *Machine Learning*, 7, 195-225.

Elman, J. L. (1993). Learning and Development in Neural Networks: The Importance of Starting Small. *Cognition*, 48, 71-99.

Gaudiano, P. & Grossberg, S. (1991). Vector Associative Maps: Unsupervised Real-Time Error-Based Learning and Control of Movement Trajectories. *Neural Networks*, 4, 147-183.

Goldberg, D.E. (1989) *Genetic Algorithms in Search Optimization and Machine Learning*. Addison Wesley,

Grinasty, M., Tsodyks, M. V., & Amit, D. J. (1993). Conversion of Temporal Correlations Between Stimuli to Spatial Correlations Between Attractors. *Neural Computation*, 5, 1-17.

Happel, B. L. M. & Murre, J. M. J. (1994). Evolving Complex Dynamics in Modular Interactive Neural Networks. In Press

Horne, B. G. & Giles, C. L. (1994). *An Experimental Comparision of Recurrent Neural Networks* NEC Research Institute.

Jordan, M. I. (1986). *Serial Order: A Parallel Distributed Processing Approach* (Tech Report No. 8604). University of California Institute for Cognitive Science.

Kalman, B. L. & Kwasny, S. C. (1994). *TRAINREC: A System for Training Feedforward & Simple Recurrent Networks Efficiently and Correctly*. (Technical Report No. 94.02). P.N.P Program, Washington University in St Louis.

Manolios, P. & Fanelli, R. (1994). First-Order Recurrent Neural Networks and Deterministic Finite State Automata. *Neural Computation*, 6, 1155-1173.

McCann, P. J. & Kalman, B. L. (1994). *A Neural Network Model for the Gold Market* No. 94.09). P.N.P Program, Washington University in St Louis.

McClelland, J. L. & Elman, J. L. (1986). Interactive Processes in Speech Perception: The TRACE Model. In J. L. McClelland & D. E. Rumelhart (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Volume 2: Psychological and Biological Models*. M.I.T. Press.

Mel, B. W. (1994). Information Processing in Dendritic Trees. *Neural Computation*, 6, 1031-1085.

Metzger, Y. & Lehmann, D. (1994). Learning Temporal Sequences by Excitatory Synaptic Changes Only. *Network Computation in Neural Systems*, 5(1), 89-99.

Minsky, M. & Papert, S. (1969). *Perceptrons*. Cambridge, M.A.: M.I.T Press.

Mozer, M. C. (1993). Neural Net Architectures for Temporal Sequence Processing. In A. Weigend & N. Gershenfield (Eds.), *Predicting the Future and Understanding the Past*, Addison-Wesley.

Murre, J. M. J. (1992). *Learning and Categorization in Modular Neural Networks*. Harvester Wheatsheaf.

Noda, I. (1994). A Model of Recurrent Neural Networks that Learn State-Transitions of Finite State Transducers. In *Proceedings of World Congress on Neural Networks*, 4 (pp. 447-452). Town & Country Hotel, San Diego, California, USA.: INNS Press.

Principe, J. C., de Vries, B., & de Oliveira, P. G. (1993). The Gamma Filter-A New Class of Adaptive IIR Filters with Restrtricted Feedback. *IEEE Transactions on Signal Processing*, 41, 649-656.

Principe, J. C. & Turner, L. A. (1994). Word Spotting with the Gamma Neural Model. In *Proceedings of World Congress on Neural Networks*, 4 (pp. 502-505). Town & Country Hotel, San Diego, California, USA.: Lawrence Earlbaum Associates.

Rall, W. (1964). Theoretical Significance of Dendritic Trees for Neuronal Input-Output Relations. In R. Reiss (Eds.), *Neuronal Theory and Modeling* Stanford University Press.

Robinson, T., Hochberg, M., & Renals, S. (1995). The use of Recurrent Neural Networks in Continuous Speech Recognition. In Press

Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning Internal Representations by Error Propagation. In D. E. Rumelhart & J. L. McClelland (Eds.), *Parallel Distributed Processing Explorations in the Microstructure of Cognition Volume 1: Foundations* M.I.T. Press.

Rumelhart, D. E. & McClelland, J. L. (Ed.). (1986). *Parallel Distributed Processing Explorations in the Microstructure of Cognition Volume 1: Foundations*. M.I.T Press.

Schmidhuber, J. (1992). A Fixed Size Storage  $O(n^3)$  Time Complexity Learning Algorithm for Fully Recurrent Continually Running Networks. *Neural Computation*, 4, 243-248.

Schwartz, J. T. (1989). The New Connectionism: Developing Relationships Between Neuroscience and Artificial Intelligence. In S. R. Graubard (Eds.), *The Artificial Intelligence Debate: False Starts Real Foundations* (pp. 123-141). M.I.T Press.

Servan-Schreiber, D., Cleeremans, A., & McClelland, J. L. (1991). Graded State Machines: The Representation of Temporal Contingencies in Simple Recurrent Networks. *Machine Learning*, 7, 161-193.

- Stevens, C. F. (1989). How Cortical Interconnectedness Varies With Network Size. *Neural Computation*, **1**, 473-479.
- Surkan, A. J. & Skurikhin, A. N. (1994). Memory Neural Networks Applied to the Prediction of Daily Energy Usage. In *Proceedings of World Congress on Neural Networks*, 2 (pp. 254-259). Town and Country Hotel, San Diego, California, USA.: INNS Press.
- Tsoi, A. C. & Back, A. D. (1994). Locally Recurrent Globally Feedforward Networks: A Critical Review of Architectures. *IEEE Transactions on Neural Networks*, **5**(2), 229-239.
- Wan, E. A. (1993). Discrete Time Neural Networks. *Journal of Applied Intelligence*, **3**, 91-105.
- Werbos, P. J. (1990). Backpropagation Through Time: What it Does and How to do it. *Proceedings of the IEEE*, **78**, 1550-1560.
- Williams, R. J. & Zipser, D. (1989). Experimental Analysis of the Real-time Recurrent Learning Algorithm. *Connection Science*, **1**, 87-111.
- Williams, R. J. & Zipser, D. (1989). A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. *Neural Computation*, **1**, 270-280.
- Zipser, D. (1989). A Subgrouping Strategy that Reduces Complexity and Speeds up Learning in Recurrent Networks. *Neural Computation*, **1**, 552-558.



## Appendices

Although the research described in this thesis was performed using the Neuralworks simulator, some source code still had to be written in order to create RTRL and Gamma networks. The learning and summation functions created are presented here. All code was written in C and uses structures found in the package used in conjunction with Neuralworks, called User Defined Neuro Dynamics. Comments about the code can be found within the body of code itself and are bracketed with `/* */` in the standard C format.

## Appendix 1: The Gamma Model Learning Rule

The following code implements the gamma learning rule described in equations 2.49 and 2.50

```
/* this is the learning rule for the Gamma memory algorithm */

#ifdef(ANSI_HEADER)

NINT um_l_gamma( USR_PE *upep, USR_CN_HDR *uchp, USR_LYR *ulp)

#else

NINT um_l_gamma(upep, uchp, ulp)
USR_PE *upep;           /* pointer to current PE */
USR_CN_HDR *uchp;       /* pointer to connection header */
USR_LYR *ulp;           /* pointer to current layer i.e one above */
/*
                        /* gamma layer */

#endif

{
    USR_PE *firsti;      /* pointer to first PE of i group */
    USR_PE *curpe;       /* pointer to offset to conn table for first i */
    USR_PE *kpe;         /* pointer to PE for alpha calculation */
    USR_PE *ppe;         /* pointer to previous PE for alpha calculation */
    USR_CONN *connection; /* pointer to connection(s) from gamma layer */
    USR_CONN *ct;        /* pointer to connection table (for backprop */
                        /* calculation) */
    USR_CN_HDR *header;  /* pointer to connection header at gamma + 1 */

    REAL totsum, ksum, alpha, oldalpha, deltai, lcoef, wv;
    NINT gape, ker, connoff, outnodes, k, i, inp, inputs, sizewts, wx;

    if ( IS_POSTLYR ) {
        return (0);      /* do little on postlayer call */
    }                    /* end of if (IS_POSTLYR) statement */

    if ( IS_INIT ) return (0); /* do nothing on init (handled by Neuralware) */
    /*

                        /* IS_PRELYR does all the work! */

    if ( IS_PRELYR ) {

        first_time_flag = 1; /* set first_time_flag to TRUE to force */
                        /* PRELYR call */

        inputs = ulp->l_prev->l_prev->num_pes;
        k = (ulp->l_prev->num_pes / inputs) - 1; /* calculates kernel size */
        firsti = ulp->l_prev->l_ep; /* first node in gamma layer */
        sizewts = ulp->functions.size_wts; /* size to skip connections in hidden layer */
    }
}
```

```

/* this is the i loop, which loops through the input lines */
for (i=0; i < inputs; i++){

    totsum = 0;          /* accumulator for total error (output layer) refers to m loop */
    curpe = ulp->l_ep;    /* initialise pointer to first PE in hidden layer */
    connoff = i * (k+1); /* offset to conn table for firsti */

    /* we are now at the m loop, which loops through the nodes in the output layer */

    for (outnodes = 0; outnodes < ulp->num_pes; outnodes++, curpe = curpe-
    >pe_next) {
        kpe = firsti;          /* pick up first node in ith kernel */
        ksum = 0;              /* accumulator for delta mu calculation */
        header = curpe->io_wtOFF; /* connection header for current (m) hidden layer*/
                                /* node */
        connection = &header->conn_table[0]; /* first connection for this npde */

        UPDWXP(connection, sizewts); /* to skip bias MAKE SURE IT'S
        /* CONNECTED!!! */

    /* the following loop skips through the gamma layer connections of the current */
    /* (m) node skipping connections to previous gamma kernels */

        for (wx = 0; wx < connoff; UPDWXP(connection, sizewts), wx++);

    /* now we start the k loop, which calculates a separate value for all m */
    /* note that gape = 1 is starting point because alpha of first PE in kernel */
    /* is always = 0 so we skip connections and alpha for k=0 */

        for (gape = 1; gape <= k; gape++) {
            UPDWXP(connection, sizewts); /* step through k connections */
            kpe = kpe->pe_next;          /* skip to next node in kernel */
            ksum += connection->weight * kpe->des_val;
        } /* end of k loop */

        totsum += curpe->err_val * ksum; /* e * f taken from err_val via control
strategy
    } /* end of m loop */

    deltai = LCOEF3 * totsum;

    /* now we can do the alpha calculation for each PE in the gamma layer */
    /* checking where each kernel begins. this value is to be used in the */
    /* next pass. The alpha for the first unit in a kernel = 0 remember to */
    /* store kernel size in LOCEF3 mu is stored in kpe->errfac alpha is stored */
    /* in kpe->des_val */

    oldalpha = 0.0; /* set up for kernel calculation */
    ppe = firsti;   /* first node in kernel */
    kpe = ppe->pe_next; /* second node in kernel */

    for (inp = 0; inp <= k; inp++) {
        firsti->err_fac += deltai; /* update mu, stored for each unit in */
                                /* gamma layer */

        if (firsti->err_fac < 0.0) firsti->err_fac = 0.0;
        if (firsti->err_fac > 2.0) firsti->err_fac = 2.0;
    }

```

```

    firsti = firsti->pe_next;          /* skip first i on, leave pointing at first */
                                        /* node of next kernel */
}                                       /* end of inp loop */

for (ker = 1; ker <= k; ker++) {
    alpha = (((1 - kpe->err_fac) * kpe->des_val) + (kpe->err_fac * oldalpha)) +
            (ppe->out_val - kpe->out_val);
    oldalpha = kpe->des_val;
    kpe->des_val = alpha;               /* store for next set of calculations */
    ppe = kpe;                         /* move along to next units in kernel */
    kpe = kpe->pe_next;
}                                       /* end of k loop */

}                                       /* end of i loop */
return(0);
}                                       /* end of PRELYR processing */

/* the following is done for every PE! This is normal processing */

if (first_time_flag) {
    ulp->pe_kcur = upep;               /* this forces a postlayer call (needed for updates!) */
    first_time_flag = 0;
}                                       /* end of if (first_time_flag) statement */

/* This code calculates the backprop learn and weight update for the hidden layer */
/* and has been lifted directly from usermath.c */

lcoef = LCOEF1 * upep->err_val;

sizewts = ulp->functions.size_wts;

for ( wx = 0, ct = &uchp->conn_table[0];
      wx < uchp->num_conns; wx++, UPDWXP(ct,sizewts) ) {
    if ( (ct->flag & (CN_DISABLED|CN_WT_MASK)) != CN_VAR )
        continue;                    /* Not a variable weight */
                                        /* compute the weight change */
    wv = lcoef * ct->src_pe->out_val + LCOEF2 * ct->last_dw;
    ct->last_dw = wv;
    ct->weight += wv;
}

/* end of normal processing */
return(0);
}

```

## Appendix 2: The Summation Function for Gamma Kernels

This code is to simulate the summation function for units in a gamma kernel described in equations 2.47 and 2.48

```
/* this is the summation function for the gamma layer */
/* it is largely based on the summation function given in usermath.c */
/* with the addition of an additional if / else if function at the */
/* end of the code, which represents equations 41 and 42 in the */
/* De Varies and Principe paper */

#ifdef ANSI_HEADER

NINT um_s_gamma( USR_PE *upeg, USR_CN_HDR *uchp, USR_LYR *ulp)

#else

NINT um_s_gamma(upeg, uchp, ulp)
    USR_PE *upeg;                /* pointer to current PE */
    USR_CN_HDR *uchp;            /* pointer to connection header */
    USR_LYR *ulp;                /* pointer to current layer */

#endif

{
    NINT wx, wf, sizewts;
    REAL accum;
    USR_CONN *ct;                /* connection table for current PE */

    if ( IS_INIT )
        return (0);

    if (IS_PRE_POST)
        return (0);

    if (uchp->num_conns == 1) {
        ct = &uchp->conn_table[0];
        upeg->sum_val = ct->src_pe->out_val;
    } else {
        accum = 0.0;
        sizewts = ulp->functions.size_wts;
        for (wx = 0, ct = &uchp->conn_table[0];
            wx < uchp->num_conns; wx++, UPDWXP(ct,sizewts) ) {
            wf = ct->flag;
            if ( (wf & CN_DISABLED) != 0) continue;
            if (upeg == ct->src_pe)
                accum += (ct->src_pe->trn_val * (1 - upeg->err_fac));
            else
                accum += (ct->src_pe->trn_val * upeg->err_fac);
        }
        upeg->sum_val = accum;
    }

    /* end of if (uchp->num_conns == 1) statement */
}
return(0);
```



### Appendix 3: The Original RTRL Algorithm

The following piece of code implements the original form of the RTRL algorithm as devised by Williams and Zipser (1989) and as described in section 2.1.7 of the literature review.

```
/* this is the learning rule for the RTRL algorithm */

#ifdef(ANSI_HEADER)

NINT um_l_rtrl( USR_PE *upep, USR_CN_HDR *uchp, USR_LYR *ulp)

#else

NINT um_l_rtrl(upep, uchp, ulp)
USR_PE *upep; /* pointer to current PE */
USR_CN_HDR *uchp; /* pointer to connection header */
USR_LYR *ulp; /* pointer to current layer */

#endif

{
    USR_PE *pel; /* pointer to other nodes for P(i,j,k) reference */
    USR_CN_HDR *wap2; /* pointer to weights table for above node */
    USR_CONN *ctw; /* pointer to connections table in above Header */
    USR_CONN *ctp; /* pointer to incoming connections for this node */

    NINT sizewts, wx, othersn, iw, rtrlwts; /* mainly counters except sizewts */
    REAL sum, errsum; /* accumulators, one for P(i,j,k) calculation, other for weight */

    if ( IS_PRELYR ) {
        first_time_flag = 1; /* set first_time_flag to TRUE to force POSTLYR call */
        return (0); /* do little on prelayer call */
    } /* end of if (IS_PRELYR) statement */

    if ( IS_INIT ) return (0); /* do nothing on init (handled by Neuralware) */

    /* IS_POSTLYR updates the P(ijk) with the newly calculated ones */

    sizewts = ulp->functions.size_wts; /* for UPDWXP (moves pointer over */
    /* connections) */

    if ( IS_POSTLYR ) {
        pel = ulp->l_ep; /* first PE (node) in layer ulp (this RTRL layer) */

        for ( othersn = 0; othersn < ulp->num_pes; othersn++, pel = pel->pe_next ) {
            wap2 = pel->io_wtoff; /* collect CN_HDR of the other node */
            ctw = &wap2->conn_table[0]; /* ctw points to its connections */

            for ( wx = 0 ;
                wx < wap2->num_conns; wx++, UPDWXP(ctw, sizewts)) {
```

```

        ctw->weights[0] += ctw->weights[1];          /* update weights */
        for (rtrlwts = 0; rtrlwts < ulp->num_pes; rtrlwts++) {
            ctw->weights[2 + (rtrlwts * 2)] = ctw->weights[3 + (rtrlwts * 2)];
        }
        /* end of rtrlwts (P(i,j,k) loop */
        /* end of wx (connections) loop */
    }
    /* end of others (nodes in RTRL layer) loop */
    return (0);
}
/* end of if IS_POSTLYR statement */

if (first_time_flag) {
    ulp->pe_kcur = upep;          /* this forces a postlayer call (needed for updates!) */
    first_time_flag = 0;
}
/* end of if (first_time_flag) statement */

/* START OF MAIN CODE (but note sizewts set up above)
/* Neuralware provides the i-loop, looping over all RTRL nodes calling this routine*/
/* once for each one */

/* j-loop, loop over all incoming connections to this PE (node)-recurrent */
/* and upwards */
/* ctp points to the connections */

for (wx = 0, ctp = &uchp->conn_table[0];
    wx < uchp->num_conns; wx++, UPDWXP(ctp, sizewts)) {

    pel = ulp->l_ep;          /* first PE (node) in layer ulp (this RTRL layer) */
    errsum = 0;              /* accumulator for weight update sum (at end of j loop) */

    /* k-loop, loop over all other RTRL nodes. pel starts at the first and */
    /* skips down through the linked list (pe_next) */

    for (others = 0; others < ulp->num_pes; others++, pel = pel->pe_next) {
        wap2 = pel->io_wtoff;          /* collect CN_HDR of the other node */
        ctw = &wap2->conn_table[0];   /* ctw points to its connections */
        sum = 0;                      /* accumulator for P(ijk) calculation */

        /* skip the input weights (l-loop is over RTRL nodes only) */
        /* skipping by looking for a connection FROM a node in the same layer */

        while (ctw->src_pe->io_layerx != ulp->l_selfx) UPDWXP(ctw, sizewts);

        /* l-loop, over RTRL nodes, sum of weights(k,l) * P(i,j,l) */
        /* P(i,j,k) are stored after weight and delta-weight (->weights[0] and [1]) */
        /* Old P(i,j,k) are in even no:s (2,4,6) and New P(i,j,k) in odd no:s (3,5,7) */

        for (rtrlwts = 0; rtrlwts < ulp->num_pes; rtrlwts++) {
            sum += ctw->weights[0] * ctp->weights[2 + (rtrlwts * 2)];
            UPDWXP(ctw, sizewts);
        }
        /* end of rtrlwts l-loop */

        /* out_val should be the old y (output) value: make sure the control file agrees! */

        if (pel==upep)
            sum += ctp->src_pe->out_val;          /* if i = j */
            /* kronecka's delta calculation */

        /* Final calculation of New P(i,j,k) into odd no: wts */
        /* trn_val is NEW y (y(t+1)) */

```



```

        ctp->weights[3 + (others * 2)] = pel->tm_val * (1 - pel->tm_val) * sum;

/* LEARNING CALCULATION */
/* This is the weight update summation calculation: doesn't have anything to */
/* do with the above calculation for P(i,j,k), but USES New P(i,j,k) */
/* We are accumulating errors(k) * OLDP(i,j,k) */
/* pel refers to PE(k), ctp->weights to OldP(i,j) (and index in for k) */

        errsum += pel->err_val * ctp->weights[2 + (others * 2)];
    }
    /* end of others k-loop */

    ctp->weights[1] = LCOEF1 * errsum;
    /* Delta weights (i,j) final calculation */

}
/* end of wx (j) loop */

return(0);
}

```

#### Appendix 4: A Modified form of the RTRL algorithm, for use with networks with sparse connectivity.

The following piece of code was used for the experiments described in chapter four in section 4.1

```

/* this is the learning rule for the RTRL algorithm used in chapter four */

#ifdef(ANSI_HEADER)

NINT um_l_rtrl( USR_PE *upeg, USR_CN_HDR *uchp, USR_LYR *ulp)

#else

NINT um_l_rtrl(upeg, uchp, ulp)
USR_PE *upeg; /* pointer to current PE */
USR_CN_HDR *uchp; /* pointer to connection header */
USR_LYR *ulp; /* pointer to current layer */

#endif

{
    USR_PE *pel; /* pointer to other nodes for P(i,j,k) reference */
    /*
    USR_CN_HDR *wap2; /* pointer to weights table for above node */
    USR_CONN *ctw; /* pointer to connections table in above Header */
    USR_CONN *ctp; /* pointer to incoming connections for this node */

    NINT sizewts, wx, otherns, iw, rtlwts; /* mainly counters except sizewts */
    REAL sum, errsum; /* accumulators, one for P(i,j,k) calculation, other for weight */
    /*

    if ( IS_PRELYR ) {
        first_time_flag = 1; /* set first_time_flag to TRUE to force POSTLYR call */
        return (0); /* do little on prelayer call */
    } /* end of if (IS_PRELYR) statement */

    if ( IS_INIT ) return (0); /* do nothing on init (handled by Neuralware) */

    /* IS_POSTLYR updates the P(ijk) with the newly calculated ones */

    sizewts = ulp->functions.size_wts; /* for UPDWXP (moves pointer over
                                         connections) */

    if ( IS_POSTLYR ) {
        pel = ulp->l_ep; /* first PE (node) in layer ulp (this RTRL layer) */

        for (otherns = 0; otherns < ulp->num_pes; otherns++, pel = pel->pe_next) {
            wap2 = pel->io_wtoff; /* collect CN_HDR of the other node */
            ctw = &wap2->conn_table[0]; /* ctw points to its connections */

            for (wx = 0 ;

```

```

wx < wap2->num_conns; wx++, UPDWXP(ctw, sizewts)) {

    ctw->weights[0] += ctw->weights[1];      /* update weights */
    for (rtrlwts = 0; rtrlwts < ulp->num_pes; rtrlwts++) {
        ctw->weights[2 + (rtrlwts * 2)] = ctw->weights[3 + (rtrlwts * 2)];
    }
    /* end of rtrlwts (P(i,j,k) loop */
    /* end of wx (connections) loop */
    /* end of othersns (nodes in RTRL layer) loop */
}
first_time_flag = 0;
return (0);
}
/* end of if IS_POSTLYR statement */

if (first_time_flag == 2) return(0);
if (first_time_flag == 1) {
    ulp->pe_kcur = upep; /* this forces a postlayer call (needed for updates!) */
    first_time_flag = 0;
}
/* end of if (first_time_flag) statement */

/* START OF MAIN CODE (but note sizewts set up above) */
/* Neuralware provides the i-loop, looping over all RTRL nodes calling this routine */
/* once for each one */

/* j-loop, loop over all incoming connections to this PE (node)-recurrent and
upwards */
/* ctp points to the connections */

for (wx = 0, ctp = &uchp -> conn_table[0];
    wx < uchp->num_conns; wx++, UPDWXP(ctp, sizewts)) {

    pel = ulp->l_ep; /* first PE (node) in layer ulp (this RTRL layer) */
    errsum = 0; /* accumulator for weight update sum (at end of j loop) */

    /* k-loop, loop over all other RTRL nodes. pel starts at the first and */
    /* skips down through the linked list (pe_next) */

    for (othersns = 0; othersns < ulp->num_pes; othersns++, pel = pel->pe_next) {
        wap2 = pel->io_wtoff; /* collect CN_HDR of the other node */
        ctw = &wap2->conn_table[0]; /* ctw points to its connections */
        sum = 0; /* accumulator for P(ijk) calculation */

        /* skip the input weights (l-loop is over RTRL nodes only) */
        /* skipping by looking for a connection FROM a node in the same layer */

        while ( ctw->src_pe->io_layerx != ulp->l_selfx) UPDWXP(ctw, sizewts);

        /* l-loop, over RTRL nodes, sum of weights(k,l) * P(i,j,l) */
        /* P(i,j,k) are stored after weight and delta-weight (->weights[0] and [1]) */
        /* Old P(i,j,k) are in even no:s (2,4,6) and New P(i,j,k) in odd no:s (3,5,7) */

        for (rtrlwts = 0; rtrlwts < ulp->num_pes; rtrlwts++) {
            sum += ctw->weights[0] * ctp->weights[2 + (rtrlwts * 2)];
            UPDWXP(ctw, sizewts);
        }
        /* end of rtrlwts l-loop */

        /* out_val should be the old y (output) value: make sure the control file agrees! */

        if (pel==upep) /* if i = j */
            sum += ctp->src_pe->out_val; /* kronecka's delta calculation */
    }
}

```

```

/* Final calculation of New P(i,j,k) into odd no: wts */
/* trn_val is NEW y (y(t+1)) */

    ctp->weights[3 + (others * 2)] = pel->trn_val * (1 - pel->trn_val) * sum;

/* LEARNING CALCULATION */
/* This is the weight update summation calculation: doesn't have anything to */
/* do with the above calculation for P(i,j,k), but USES New P(i,j,k) */
/* We are accumulating errors(k) * OLDP(i,j,k) */
/* pel refers to PE(k), ctp->weights to OldP(i,j) (and index in for k) */

    errsum += pel->err_val * ctp->weights[2 + (others * 2)];

    }                                     /* end of others k-loop */

    ctp->weights[1] = LCOEF1 * errsum;    /* Delta weights (i,j) final calculation */

    }                                     /* end of wx (j) loop */

return(0);
}

```